Functional Declarative Language Design and Predicate Calculus: A Practical Approach

RAYMOND BOUTE
INTEC, Ghent University, Belgium

In programming language and software engineering, the main mathematical tool is *de facto* some form of predicate logic. Yet, as elsewhere in applied mathematics, it is used mostly far below its potential, due to its traditional formulation as just a topic in logic instead of a calculus for everyday practical use.

The proposed alternative combines a language of utmost simplicity (four constructs only) that is devoid of the defects of common mathematical conventions, with a set of convenient calculation rules that is sufficiently comprehensive to make it practical for everyday use in most (if not all) domains of interest.

Its main elements are a functional predicate calculus and concrete generic functionals. The first supports formal calculation with quantifiers with the same fluency as with derivatives and integrals in classical applied mathematics and engineering. The second achieves the same for calculating with functionals, including smooth transition between pointwise and point-free expression.

The extensive collection of examples pertains mainly to software specification, language semantics and its mathematical basis, program calculation etc., but occasionally shows wider applicability throughout applied mathematics and engineering. Often it illustrates how formal reasoning guided by the shape of the expressions is an instrument for discovery and expanding intuition, or highlights design opportunities in declarative and (functional) programming languages.

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specifications; D.3.1 [**Programming Languages**]: Formal Definition and Theory; E.1 [**Data Structures**]; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic; F.4.3 [**Mathematical Logic and Formal Languages**]: Formal Languages; I.1.1 [**Symbolic and Algebraic Manipulation**]: Expressions and Their Representation; I.1.3 [**Symbolic and Algebraic Manipulation**]: Languages and Systems; K.3.2 [**Computers and Eduction**]: Computer and Information Science Education

General Terms: Design, Documentation, Languages, Theory, Verification

Additional Key Words and Phrases: Analysis, binary algebra, calculational reasoning, databases, declarative languages, elastic operators, function equality, functional predicate calculus, generic functionals, Leibniz's principle, limits, programming languages, program semantics, quantifiers, recursion, software engineering, summation

Author's address: INTEC Department of Information Technology, Ghent University, Sint-Pietersnieuwstraat 41, B-9000 Ghent, Belgium; email: raymond.boute@intec.Ugent.be.

1. INTRODUCTION

Motivation (a): *Declarative Expression.* Automation seemingly has reduced programming to menu selection and mouse clicking. Yet, such Nintendo-style environments have severe limitations. They provide no adequate way to express formal *specification* (which needs abstraction) and restrict design models to those supported by the tools. This is especially inconvenient for hybrid systems [Alur et al. 1996; Buck et al. 1994; Vaandrager and van Schuppen 1999], for example, by imposing an algorithmic view on systems that are better modelled by equations [Boute 1991], as in physics.

So, in reality, the main design issues have just shifted to a higher abstraction level. Programming is not only writing programs to instruct computers (*direct programming*), but also specifying and reasoning about the task in the problem domain (which may be anything) and in the implementation domain (languages, semantics). Languages covering all these *indirect programming* issues must be *declarative*, which means at least: able to express *what* is required, rather than *how* to realize it.

Definitions of declarativity often come with the warning [Illingworth et al. 1989; Rechenberg 1990] that languages for direct programming, including functional and logic ones, fall short of this criterion. For instance, to express *sorting*, such languages require choosing an algorithm, so they are essentially *algorithmic*. Such overspecification can be avoided in declarative formalisms.

As argued in Rushby et al. [1998], declarative languages should not be restricted by executability concerns but must also include quantification, in a practical, convenient form [Gries 1996a; Parnas 1993].

Motivation (b): *Practical Calculation Rules.* Beside software engineering [Page 2000], programming languages constitute the main de facto practical application area of predicate logic, namely, in describing and analyzing the various language-oriented formalisms and theories [Dijkstra and Scholten 1990; Gries and Schneider 1993; Meyer 1991; Tennent 1991; Winskel 1993]. Yet, with few exceptions [Dijkstra and Scholten 1990; Gries and Schneider 1993; Parnas 1993], even there the logic is not really used as a *calculus*, but loosely, just for descriptive purposes.

In proofs and derivations, quantified formulas are handled on a "see what I mean" basis, making quantifiers mere abbreviations for "there exists" and "for all" in natural language. Taylor [2000] calls this *syncopation* and rightly observes that it often obscures the logical structure of the arguments in other areas of mathematics as well. Thus, predicate logic is used far below its potential as a practical intellectual tool to assist reasoning [Gries 1996a]. Indeed, formal calculation, guided by the shape of the formulas, has the invaluable benefit of letting the symbols do the work (*"Ut faciant opus signa"* [Boiten and Möller 2002]).

Of course, automated tools [Rushby et al. 1998; Paulson 2001] are also formal. However, it is at least as rewarding to exploit formality in formalisms for *human* communication and reasoning, since the "parallel" syntactic intuition fostered by formal calculation proves extremely useful, especially when exploring (new) areas where traditional semantic intuition is clueless, uncertain, or

still in the development stage. This is tacitly taken for granted in classical applied mathematics, for example, (hand) calculation with derivatives and integrals is essentially formal and trouble-free, but in other areas similar fluency is rare.

Indeed, common mathematical conventions are strong in Algebra and Analysis (e.g., rules for $\int$ in every introductory Analysis text), weaker in Discrete Mathematics (e.g., rules for $\sum$ only in very few texts [Graham et al. 1994]), and poor in Predicate Logic (e.g., disparate conventions for $\forall$ and $\exists$, rules in most logic texts not suited for practice). These relative strengths are inversely proportional to the needs in CS and programming.

The main cause is insufficient attention to calculation rules. Even excellent texts on logic [Cori and Lascar 2000; Mendelson 1987] present predicate logic only as a proof- and model-theoretic topic, and with rules so impractical that the effort is abandoned after the first few chapters, reverting to an informal style for the remainder. Using predicate logic formally in everyday practice requires a *calculus* with a comprehensive collection of rules [Gries and Schneider 1993] capturing recurrent patterns that otherwise have to be spelled out in detail every time.

There is considerable agreement [Dijkstra 1992; Gries and Schneider 1993; Parnas 1993] that in any language design to solve the unsatisfactory state of affairs, the *calculation rules* must be the prime criterion, to the extent that *calculations can be guided by the shape of the expressions*.

Relation to Other Approaches. To achieve these goals, some authors advocate total redesign [Dean and Hinchey 1996; Dijkstra and Scholten 1990; Gries 1996a], even at the cost of compatibility with the rest of mathematics.

Our design has the same priorities, yet salvages common conventions better than one might expect. Orthogonal combination of just four constructs avoids the defects, yet adds simple formal calculation rules and new, surprisingly flexible forms of expression.

The approach to predicate calculus is *practical* in many ways. It extends the concept of functionals familiar from applied mathematics and engineering, and embeds logic in mathematics as arithmetic restricted to $\{0, 1\}$. Its focus is designing rules needed for actual use rather than the technicalities expounded in logic texts. Quantifiers $\forall$ and $\exists$ are predicates over predicates, extend $\wedge$ and $\vee$ in a simpler way than $\sum$ extends $+$, and have clear rules. This is very close to Leibniz's ideal of a true logic calculus [Dijkstra 2000], invalidating the prejudice that formal predicate calculus is unavoidably tedious.

The resulting reasoning style is *calculational* [Gries and Schneider 1993], that is, by expression manipulation and chaining formulas as in elementary algebra and analysis. The steps are justified by given axioms or derived theorems and chained by transitive relations, rather than via sequent and inference rules. Proofs and derivations are thereby more linear than with the traditional tree structure.

The most similar predicate calculus in the literature is found in Gries and Schneider [1993]. Ours differs in the following ways: (i) it supports both pointwise and point-free formulations, (ii) it is function- rather than

expression-centered, (iii) elastic operators replace ad hoc abstractors, (iv) the design principle extends beyond associative and commutative operators, (v) full handling of types (also empty domains) is embedded in the calculation rules, (vi) derivation starts from fewer axioms (function equality).

Overview. Section 1 outlines the language requirements, a simple design that fulfils them, the principle of calculational reasoning, and *generic functionals* supporting the sequel. Section 2 presents a *functional predicate calculus*, axiomatizes quantifiers, calculationally derives some typical rules, and lists various others. A first batch of applications in Section 3 illustrates how defects in common conventions are corrected, how certain oversights in other calculi are avoided, and how reasoning guided by the shape of the expressions becomes an instrument for discovery expanding intuition.

The second part of the article shows how the same language design covers a much wider application range. Section 4 explains the underlying language pragmatics and the *functional mathematics* principle used in designing *elastic* operators and additional generic functionals. Section 5 provides more specific examples in formal semantics, program transformation, program correctness and a minitheory of recursion obviating partial functions. Examples in the "continuous" world allow to conclude that, just as Gries' calculational logic is the "glue" between topics in discrete mathematics, our functional calculus is the glue between discrete and continuous applied mathematics.

The relation to mechanized tools, proof style and related issues is briefly discussed.

NOTE. Common programming languages are illustrated by programs, declarative ones by formulas *and proofs*. So this article will contain many proofs, illustrating issues ranging from style to concepts and applications. To reduce its length, we mostly (but not always) restrict application-related proofs to programming topics, and minimize derivations of the rules themselves, referring to Boute [2004] for a more extensive treatment.

2. DECLARATIVE FORMALISM DESIGN

A *formalism* is a language (notation) together with formal manipulation rules. The merits of the language are measured not only by its scope and expressiveness but even more by the degree to which it supports formal rules assisting human reasoning.

2.1 Language Requirements versus Common Conventions

2.1.1 *Styles of Expression.* Mathematical expression can be *pointwise*, using variables (or dummies), or *point-free*, without dummies. Common mathematics is mostly pointwise.

Combinator calculus [Barendregt 1984], Tarski's set theory [Tarski and Givant 1987], and Backus' FP [Backus 1978] are point-free variants of lambda calculus, set theory and (functional) programming, respectively. Point-free styles are more compact and yield rules with an appealing algebraic flavor.

Yet, as illustrated in later application examples, formalisms meant for practical use should support *both* styles, including smooth transformation rules between them.

2.1.2 *Avoiding Defects.* Even if we needed only pointwise styles, common conventions fall short because of their many peculiarities, ambiguities, incompatibilities and imprecise definitions hampering formal reasoning. The typical excuses that they are "just a matter of notation" or that "we have learned how to be careful" are counterproductive, since having to be on guard precludes letting the symbols do the work. Notational flaws are never "just a matter of notation" but symptoms of deeper conceptual defects; to paraphrase Boileau: good concepts induce clear notation. So, only overall conceptual design, not ad hoc patching, can avoid all problems. For concreteness, we give a few illustrations of common defects in general formalism design and in operator design.

(a) Binding conventions for variables (dummies) are sloppy, ambiguous, and require context information (precluding clear formal rules). Let us show how pervasive this is.

Dummies in expressions like $\sum_{i<j} f(i,j)$ are bound "by suggestion" via the context: $a = \sum_{i<j} f(i,j)$ suggests sum over i and j (sure?), but $a_j = \sum_{i<j} f(i,j)$ over i only.

A typical defect is abusing the set membership relation $\in$ for binding a dummy. Ubiquitous patterns are (i) $\{x \in X \mid p\}$ with p Boolean, and (ii) $\{e \mid x \in X\}$ with any e, as in $\{m \in \mathbb{Z} \mid m < n\}$ and $\{n \cdot m \mid m \in \mathbb{Z}\}$. The ambiguity is revealed by taking $y \in Y$ for p and e. Example: If $Even = \{2 \cdot m \mid m \in \mathbb{Z}\}$, then $\{n \in \mathbb{Z} \mid n \in Even\}$ and $\{n \in Even \mid n \in \mathbb{Z}\}$ depend on choosing (i) or (ii) and which occurrence of $\in$ binds n. Ad hoc preference for (i) in this case does not remove the inherent ambiguity and restricts expressivity: what if (ii) is wanted? Such defects explain why formal calculation with expressions like $\{x \in X \mid p\}$ is rare to nonexistent in the literature.

(b) Many conventions violate a crucial criterion in formalism design, namely *Leibniz's principle*: stated informally (and formalized later), equals should always be replaceable by equals. One example is *ellipsis*, writing dots as in $a_0 + a_1 + \cdots + a_n$. By Leibniz's principle, if $a_i = i^2$ and $n = 7$, this should equal $0 + 1 + \cdots + 49$ or $49 \cdot 50/2$, whereas more likely a sum of squares (equal to 140) is intended.

(c) Universal use of notations of the form $\sum_{i=m}^{n} e$ and, more insidiously, handling them by informal interpretation ("see what I mean"), creates the illusion that they are fairly well understood. Mathematical software often contains errors as a result, but may also expose them. For instance, Pugh [1994] notes that, in *Mathematica* [Wolfram 1996]:

$$\sum_{i=1}^{n} \sum_{j=i}^{m} 1 \;=\; \frac{n \cdot (2 \cdot m - n + 1)}{2}. \tag{1}$$

In Maple, `sum(sum(1,j=i..m),i=1..n);` returns $m(n+1) + \frac{3}{2}n + \frac{1}{2} - \frac{1}{2}(n+1)^2 - m$, which is the same after simplifiction. However, correct formal calculation yields

$$\sum_{i=1}^{n} \sum_{j=i}^{m} 1 = (\textbf{if } k \geq 1 \textbf{ then } \frac{k \cdot (2 \cdot m - k + 1)}{2} \textbf{ else } 0) \textbf{ where } k := min\,(m,n). \tag{2}$$

For instance, substituting $n, m := 3, 1$ in (1) yields 0 instead of the correct sum 1.

With the growing role of mathematical software, poor conventions in many areas need attention to avoid regrettable design decisions that are hard to revoke once work is spent on implementations. Hand calculation is an even more demanding benchmark.

2.1.3 *The Design Task.* Poor language design may appear tolerable if one considers only expressivity (ignoring ambiguities, inconsistencies), but to formal calculation it is *fatal*. So, part of declarative formalism design is providing simple, trouble-free notation.

In view of the preceding discussion, this task appears huge, but in retrospect (as seen by the end of this article) a very small design proves sufficient. This is due to many lessons learned from programming language design, in particular the discipline of careful language engineering and the principle of orthogonality.

More demanding were the *pragmatics*, that is, designing a framework for the language's actual use, in particular formal calculation rules, which are the main topic of this article.

2.2 A Simple Declarative Formalism

2.2.1 *The Functional Mathematics Approach and the Function Concept. Functional mathematics* is an approach to formalism design, based on defining existing and new mathematical concepts as *functions*. This turns out most fruitful where it is not (yet) common. A preferred embodiment in concrete syntax is *Funmath* [Boute 1993a, 1993b].

From just four constructs, common conventions as well as new forms are synthesized. The first use yields expressions with familiar form, readable without knowledge of Funmath. It allows "use before/without (formal) definition", since it differs from common conventions only by being defect-free and enjoying formal calculation rules. The second use includes the point-free style, and may require prior familiarization.

Our *function* concept is the usual one, but given the variations in the literature (e.g., some include a codomain), we add a brief comment. A function is an object in its own right, not identified with its set-theoretic representation by pairs (its *graph* [Lang 1983]). By definition, it is fully specified by two attributes: a *domain* (the set on which it is defined) and its *mapping*, associating with every domain element a unique image.

2.2.2 *The Four Syntactic Constructs for Mathematical Expressions.* This is a first outline; refinements not needed for our predicate calculus are given later. Although we use the concrete Funmath syntax, the abstract syntax is easily inferred.

Given the simplicity, we freely alternate BNF with the convention as in logic texts of using metavariables for expressions, whichever is convenient. Here are the constructs.

$$\mathit{expression} \;::=\; \mathit{identifier} \mid \mathit{application} \mid \mathit{abstraction} \mid \mathit{tupling}. \tag{3}$$

Henceforth, we use metavariables d, e for arbitrary expressions, f, g, h, $\star$ for function expressions, p, q, r for Boolean expressions (propositions), X, Y, Z

for set expressions, all possibly with primes. This type distinction will become obvious soon. Metavariables for identifiers or tuples of identifiers are: i for any kind and u, v for variables.

(1) *Identifier.* An identifier can be any symbol or string except binding colon, filter mark, abstraction dot, parentheses, and a few boldface keywords (given as we proceed).

Identifiers are *introduced* (declared) by *bindings* of the form $i : X \mathrel{\mathbin{\wedge}} p$, read "$i$ in X satisfying p". Identifiers in i should not occur free in expression X. The *filter* $\mathrel{\mathbin{\wedge}} p$ (or **with** p) is optional, for example, $n : \mathbb{N}$ and $n : \mathbb{Z} \mathrel{\mathbin{\wedge}} n \geq 0$ are interchangeable. Syntactic sugar: $i := e$ stands for $i : \iota\, e$. As explained later, we write $\iota\, e$, not $\{e\}$, for singleton sets.

Identifiers can be *variables* (in an abstraction) or *constants* (declared by **def** *binding*). Well-established symbols (e.g., $\mathbb{B}$, $\Rightarrow$, $\mathbb{R}$, $+$, $\sqrt{\ }$) are seen as predefined constants.

(2) *Application.* For function f and argument e, the default is $f\, e$. Other affix conventions are specified by dashes in the operator's binding, for example, $— \star —$ for infix.

For clarity, parentheses are *never* used as operators, but only for emphasis or overruling affix conventions, for example, $(\star)(x, y) = x \star y$ if $\star$ is infix. Rules for making them optional (precedence rules) are the usual ones. Prefix always has precedence over infix. If f is a function-valued function, $f\, x\, y$ stands for $(f\, x)\, y$.

For infix $\star$, *partial application* is of the form $x \star$ or $\star\, y$, with $(x \star)\, y = x \star y = (\star\, y)\, x$. *Variadic application* is of the form $x * y * z$ etc., and is *always* defined to equal $F(x, y, z)$ for a suitably defined *elastic extension* F of $\star$. The vast ramifications will appear later.

(3) *Abstraction.* The form is $b \,.\, e$, where b is a binding and e an expression (extending after " . " as far as compatible with parentheses present). In $b \,.\, e$, occurrences of variables declared in b are *bound*, others are *free* [Barendregt 1984]. Intuitively, $v : X \mathrel{\mathbin{\wedge}} p \;.\; e$ denotes a function whose domain is the set of v in X satisfying p, and mapping v to e. Formally,

$$\text{DOMAIN AXIOM:} \quad d \in \mathcal{D}(v : X \mathrel{\mathbin{\wedge}} p \;.\; e) \equiv d \in X \wedge p[v := d] \tag{4}$$

$$\text{MAPPING AXIOM:} \quad d \in \mathcal{D}(v : X \mathrel{\mathbin{\wedge}} p \;.\; e) \Rightarrow (v : X \mathrel{\mathbin{\wedge}} p \;.\; e)\, d = e[v := d], \tag{5}$$

where $e[v := d]$ or $e[^{v}_{d}$ is e in which d is substituted for all free occurrences of v [Barendregt 1984; Gries and Schneider 1993].

A trivial example: If v is not free in e, we define $\bullet$ by $X \bullet e = v : X \;.\; e$, denoting *constant functions*. Special cases are the *empty function* $\varepsilon := \emptyset \bullet e$ (any e) and defining $\mapsto$ by $d \mapsto e = \iota\, d \bullet e$ for *one-point functions*, similar to *maplets* in Z [Spivey 1989].

Syntactic sugar: We let $e \mid b$ stand for $b \,.\, e$ and $v : X \mid p$ for $v : X \mathrel{\mathbin{\wedge}} p \;.\; v$.

Application of *elastic operators* ($\sum$, $\{—\}$, $\forall$ defined as functionals) to abstractions yields familiar expressions such as $\sum i : 0 \,..\, n \,.\, q^i$, $\{m : \mathbb{Z} \mid m < n\}$ and $\forall x : \mathbb{R} \,.\, x^2 \geq 0$.

(4) *Tupling.* The 1-dimensional form is e, e', e'' (any number of items), denoting a function with domain axiom $\mathcal{D}(e, e', e'') = \{0, 1, 2\}$ and mapping axiom $(e, e', e'')\,0 = e$ and $(e, e', e'')\,1 = e'$ and $(e, e', e'')\,2 = e''$. Matrices are 2-dimensional tuples. We justify later why the empty tuple is ε and for singleton tuples we define τ with $\tau\, e = 0 \mapsto e$.

Parentheses are *not* part of tupling, and are as optional in (m, n) as in $(m + n)$.

Remarks

(a) In economy of constructs, this resembles lambda calculus [Barendregt 1984], but the tupling construct is a crucial addition. Even though lambda calculus can simulate tuples to some extent, we explain later why that is not adequate for our purpose.

(b) The denotational semantics is the usual mathematical interpretation.

(c) Abstractions can make expressions cryptic to the uninitiated, except as arguments of elastic operators to synthesize common notation. Yet, they can also make definitions succinct: a mapping axiom like $x \in \mathcal{D}\, f \Rightarrow f\, x = e$ with f not free in e merges with the domain axiom $x \in \mathcal{D}\, f \equiv x \in X \wedge p$ into simply $f = x : X \mathrel{\mathbb{A}} p \;.\; e$.

2.2.3 *Axiomatic Semantics: Calculational Reasoning.* In reasonably well-designed formalisms such as algebra and analysis, calculations and proofs are usually presented as chained equalities of the form $e = e' = e''$ etc.

Calculational reasoning [Dijkstra and Scholten 1990; Gries and Schneider 1993] extends this to relations beyond "=" in the format

$$\begin{array}{l} e \;\; R \;\langle \text{Justification} \rangle \;\; e' \\ \quad R' \;\langle \text{Justification} \rangle' \;\; e'' \text{ etc.}, \end{array}$$

where the relational operators R, R' are mutually transitive, for instance $=$, $\leq$, etc. in arithmetic, $\equiv$, $\Rightarrow$ etc. in logic. The layout for the $\langle$justification$\rangle$s is attributed to Feyen.

A general inference rule (here *strict* in that the premiss must be a theorem) is

$$\text{INSTANTIATION:} \quad \frac{p}{p[v := e]}. \tag{6}$$

Important Remark for the Sequel: In this setting, a theorem p, say, $n \in \mathbb{Z} \Rightarrow n + 1 > n$, is interchangeable with a metatheorem $p[n := e]$ (or $p[\,{}^{n}_{e}]$), say, $e \in \mathbb{Z} \Rightarrow e + 1 > e$, the choice of formulation being a matter of emphasis, style, or mere convenience.

For equality, the rules [Gries and Schneider 1993] are reflexivity, symmetry, transitivity and

$$\text{LEIBNIZ'S PRINCIPLE:} \quad e = e' \Rightarrow d\,[\,{}^{v}_{e} = d\,[\,{}^{v}_{e'}. \tag{7}$$

For instance, $x + 3 \cdot y = \langle x = z^2 \rangle \;\; z^2 + 3 \cdot y$, where (7) is recognized by $d := v + 3 \cdot y$.

Calculational chaining is clear, synoptic, and avoids repetition. It highlights a *principal chain of reasoning* with the crucial steps. Such "linearization" used

to be slightly controversial since general deduction is tree-structured. Yet, in practice it comes rather naturally (e.g., factorizing out essential branches as lemmata). As a result, calculational proofs can be formal, more complete, more clearly structured and not longer than informal proofs, yet much shorter than traditional formal proofs [Gries and Schneider 1993; Ostroff, http://www.cs.yorku.ca/~logicE/curriculum/logic_discrete_maths.html].

2.3 Calculating with Boolean, Set and Function Expressions

2.3.1 *Boolean Equality, Operators on Booleans and Boolean Calculi.* The terms *Boolean expression* and *proposition* are used interchangeably, and the formal rules constitute *proposition calculus*. The main operators are $\neg$, $\Rightarrow$, $\equiv$, $\wedge$, $\vee$.

In this calculus, $\equiv$ is the equality sign and is used as such in calculational chaining. There are many advantages in not keeping just "$=$" but also "$\equiv$" as a separate operator [Gries and Schneider 1993], namely, reducing parentheses by giving $\equiv$ lowest precedence of all, and highlighting associativity of $\equiv$, which is not shared by $=$. We prefer "$\equiv$" over "$\Leftrightarrow$" to stress its equational character over $\Leftarrow$-cum-$\Rightarrow$. Given the associativity, we let $p \equiv q \equiv r$ stand for both $p \equiv (q \equiv r)$ and $(p \equiv q) \equiv r$, hence clearly *not* for $(p \equiv q) \wedge (q \equiv r)$.

Implication $\Rightarrow$ is not associative, but we make parentheses in $p \Rightarrow (q \Rightarrow r)$ optional, hence required in $(p \Rightarrow q) \Rightarrow r$. Its precedence is lower than $\wedge$, $\vee$ but higher than $\equiv$.

2.3.1.1 *Proposition Calculus in Practice.* For a practical calculus, a much more extensive collection of rules is required than found in classical texts on logic. Overviews can be found in Boute [2002] and Gries and Schneider [1993]. To manage this large variety, it is very helpful to have a *nomenclature* of well-chosen mnemonic names and a quick way for reconstruction or checking rules.

For instance, apart from the usual *associativity* and *commutativity* for $\wedge$ and $\vee$, we also have *monotonicity* (or *isotony*; unfortunately the nomenclature is not uniform).

$$\begin{array}{ll} \text{Right monotonocity of } \star\text{:} & (p \Rightarrow q) \Rightarrow (r \star p \Rightarrow r \star q) \\ \text{Left antimonotonocity of } \Rightarrow\text{:} & (p \Rightarrow q) \Rightarrow (q \Rightarrow r) \Rightarrow (p \Rightarrow r), \end{array}$$

where $\star$ is either $\wedge$, $\vee$ or $\Rightarrow$; the former two are also left monotonic by commutativity. These transitivity-like properties justify writing calculations in the following layouts

$$\begin{array}{l|l} \cdots \langle \text{Steps leading to } r \star p \rangle \quad r \star p & \cdots \langle \text{Steps leading to } q \Rightarrow r \rangle \quad q \Rightarrow r \\ \Rightarrow \langle \text{Justification for } p \Rightarrow q \rangle \quad r \star q & \Rightarrow \langle \text{Justification for } p \Rightarrow q \rangle \quad p \Rightarrow r. \end{array}$$

Less often found in theoretical texts is *shunting* (parentheses here for emphasis only):

$$\begin{array}{ll} \text{Shunting with } \Rightarrow\text{:} & x \Rightarrow (y \Rightarrow z) \equiv y \Rightarrow (x \Rightarrow z) \\ \text{Shunting with } \wedge\text{:} & x \Rightarrow (y \Rightarrow z) \equiv (x \wedge y) \Rightarrow z. \end{array}$$

Examples of quick shortcuts are *case analysis*: $p[_0^v, p[_1^v \vdash p$, and the many variants of Shannon's theorem, such as $p \equiv (\neg v \wedge p[_0^v) \vee (v \wedge p[_1^v)$. We call v the

Table I. Binary Algebra as a Restriction of Plain Arithmetic to 0 and 1

x, y	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0,0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0,1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1,0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1,1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
$\mathbb{B}$		$\forall$	$<$		$>$		$\not\equiv$	$\not\asymp$	$\wedge$	$\equiv$	$\rangle\!\rangle$	$\Rightarrow$	$\langle\!\langle$	$\Leftarrow$	$\vee$	
$\mathbb{R}'$			$<$		$>$		$\neq$		$\curlywedge$	$=$	$\rangle\!\rangle$	$\leq$	$\langle\!\langle$	$\geq$	$\curlyvee$	

fulcrum. These are also used in automated tautology checking, model checking and similar applications [Bryant 1992]. Rules of humanly tractable size can be verified or derived by mere head calculation. The reader can try this on $(x \Rightarrow y) \wedge (y \Rightarrow z) \equiv \neg(x \vee y) \vee (y \wedge z)$ taking fulcrum y and applying "Shannon" to the left-hand side for derivation (also using De Morgan's rule), or case analysis to the entire equality for verification.

2.3.1.2 *Binary Algebra.* Although not essential to our predicate calculus, we use propositional constants 0 and 1 rather than FALSE and TRUE. For reasons explained elsewhere [Boute 1990, 1993a] and more eloquently by [Hehner 1996], we embed binary algebra in plain arithmetic as a restriction of minimax algebra to $\{0, 1\}$ (note: Hehner uses $\{-\infty, +\infty\}$). Since this issue is also relevant to programming language design, we briefly explain.

Minimax algebra is what we call the algebra of the *least upper bound* ($\curlyvee$) and *greatest lower bound* ($\curlywedge$) operators over $\mathbb{R}' := \mathbb{R} \cup \{-\infty, +\infty\}$ [Boute 1993a]. One definition is

$$x \curlyvee y \leq z \equiv x \leq z \wedge y \leq z \quad \text{and} \quad z \leq x \curlywedge y \equiv z \leq x \wedge z \leq y. \tag{8}$$

Since $\leq$ is total, $x \curlywedge y = (y \leq x)\,?\,y \mid x$, where $p\,?\,e' \mid e$ is read as "if p then e' else e"; formally: $p\,?\,e' \mid e = (e, e')\,p$. Similarly, $z \leq x \curlyvee y \equiv z \leq x \vee z \leq y$ and its dual.

For the $\curlyvee$ and $\curlywedge$ operators alone, there exist commutativity laws, for example, $x \curlyvee y = y \curlyvee x$, associativity laws, for example, $x \curlyvee (y \curlyvee z) = (x \curlyvee y) \curlyvee z$, distributivity laws such as $x \curlyvee (y \curlywedge z) = (x \curlyvee y) \curlywedge (x \curlyvee z)$, monotonicity laws such as $x \leq y \Rightarrow x \curlyvee z \leq y \curlyvee z$ etc.

In addition, there is a rich algebra of laws combined with other arithmetic operators, such as distributivity: $x + (y \curlyvee z) = (x + y) \curlyvee (x + z)$ and $x - (y \curlyvee z) = (x - y) \curlywedge (x - z)$.

Such laws can be derived by high school algebra, using duality to save work.

Binary algebra is then the algebra of the operators $\vee$ and $\wedge$, defined in [Boute 1993a] as the restrictions of $\curlyvee$ and $\curlywedge$ to $\mathbb{B} := \{0, 1\}$. Table I, reproduced from [Boute 1993a], illustrates this for the 16 functions from $\mathbb{B}^2$ to $\mathbb{B}$ by listing $f_i(x, y)$ for $i : 0 \,..\, 15$ and $(x, y) : \mathbb{B}^2$.

All laws of minimax algebra now particularize to laws over $\mathbb{B}$, for instance:

$$x \vee y \Rightarrow z \equiv (x \Rightarrow z) \wedge (y \Rightarrow z) \quad \text{and} \quad z \Rightarrow x \wedge y \equiv (z \Rightarrow x) \wedge (z \Rightarrow y) \tag{9}$$

is just a particularization of (8) for Boolean x, y and z.

2.3.2 *Set Equality and Operators on Sets.* For handling sets in predicate calculus, we assume a modicum of calculation rules. The basic operator is $\in$. *Set equality* is defined in two parts. First, *Leibniz's principle*: $X = Y \Rightarrow (e \in X \equiv e \in Y)$ or, equivalently, $(p \Rightarrow X = Y) \Rightarrow p \Rightarrow (e \in X \equiv e \in Y)$. Second, the converse, expressed here as a strict inference rule: for new v,

$$\text{SET EXTENSIONALITY:} \quad \frac{p \Rightarrow (v \in X \equiv v \in Y)}{p \Rightarrow X = Y}. \tag{10}$$

The presence of p allows embedding set extensionality in a calculation chain as

$$\begin{array}{lll} p & \Rightarrow & \langle \text{Calculations to right-hand side} \rangle \quad v \in X \equiv v \in Y \\ & \Rightarrow & \langle \text{Set extensionality} \rangle \qquad\qquad\quad X = Y \end{array}$$

Warning: This is a deduction for $p \Rightarrow X = Y$; do *not* read $(v \in X \equiv v \in Y) \Rightarrow X = Y$. Such inference rules are used only to bootstrap the predicate calculus, not beyond.

The axioms for set operators are expressed via proposition calculus, for instance, defining $\cap$ by $x \in X \cap Y \equiv x \in X \wedge x \in Y$ and $\times$ by $(x, y) \in X \times Y \equiv x \in X \wedge y \in Y$.

The *empty* set $\emptyset$ has axiom $x \notin \emptyset$, and singletons ιe have axiom $d \in \iota e \equiv d = e$. Using Frege's *singleton injector* ι is motivated in [Forster 1992]. Our reasons to avoid { } for singletons is that it violates Leibniz's principle and that { } can be salvaged for better purposes shown later, leading to the calculation rule $e \in \{x : X \mid p\} \equiv e \in X \wedge p[^x_e$.

Not essential but convenient is letting $\mathcal{U}$ stand for the general universe, $\mathcal{F}$ for the function universe, $\mathcal{T}$ for the set universe and $\mathcal{E}$ for the universe of enumeration sets [Jensen and Wirth 1978]. In view of portability, we avoid making this specific: in some formalisms, $\mathcal{U}$ may be just the universe of discourse, in others, a general universe [Forster 1992]. Like the designers of PVS [Rushby et al. 1998], we are not overly concerned with paradoxes, but some readers may wish to consider $R := \{X : \mathcal{T} \mid \neg(X \in X)\}$, expand $R \in R$ and solve the resulting equation.

2.3.3 *Functions, Equality and Operators on Functions: Generic Functionals*

2.3.3.1 *Functions, Types, Guards.* We specify functions by a *domain axiom* and a *mapping axiom* of the form (or rewritable as) $v \in \mathcal{D}\, f \equiv p$ (or $\mathcal{D}\, f = X$) and $v \in \mathcal{D}\, f \Rightarrow q$ respectively, with $f \notin \varphi\, p$ and $v \notin \varphi\, f$. Here $\varphi\, e$ is the set of free variables in e. An example is $\mathcal{D}\, fac = \mathbb{N}$ and $fac\, 0 = 1 \wedge (n \in \mathbb{N} \Rightarrow n > 0 \Rightarrow fac\, n = n \cdot fac\, (n-1))$. When stating both axioms together, we often write q for $x \in \mathcal{D}\, f \Rightarrow q$", but "$x \in \mathcal{D}\, f \Rightarrow$" is understood, for example, defining $\sqrt{\ }$ by $x \in \mathcal{D}\, \sqrt{\ } \equiv x \in \mathbb{R} \wedge x \geq 0$ and $\sqrt{x}^2 = x \wedge \sqrt{x} \geq 0$. More refined and less verbose styles of definition will appear later.

In declarative/mathematical languages, it is sensible to identify types with sets [Lamport and Paulson 1997]. Type systems in programming languages can be seen as simplifications that ensure easy decidability (by compilers), but are often too restrictive for declarative formalisms.

Out-of-domain function applications routinely occur in nonpathological mathematical expressions [Boute 2000; Parnas 1993]. An example is

$(x \geq 0 \Rightarrow \sqrt{x}^2 = x) \wedge (x < 0 \Rightarrow \sqrt{-x}^2 = -x)$. Instantiation for nonzero x causes an out-of-domain application in one of the roots, but the corresponding antecedent ensures that the entire formula remains a theorem.

Using antecedents (or conjuncts) so that parts of a formula do not make the entire formula undefined is called *guarding* and makes it "robust" against out-of-domain applications. The approach is "portable" across formalisms, that is: largely independent of the conventions for handling "undefined" [Boute 2000; Schieder and Broy 1999]. Contrary to expectation, putting guards in theorems and function definitions turns out to be no burden at all: for theorems they just formalize the hypotheses, and for functions they make domain information "pop up" exactly when needed in formal calculation. Guarding and its ramifications are discussed extensively in [Boute 2000], but most will become evident from examples, including how refined domain specifications obviate "partial" functions.

2.3.3.2 *Function Equality.* We proceed as for sets, but with domain membership as guards.

$$\text{LEIBNIZ FOR FUNCTIONS:} \quad f = g \Rightarrow \mathcal{D}\, f = \mathcal{D}\, g \wedge (e \in \mathcal{D}\, f \cap \mathcal{D}\, g \Rightarrow f\, e = g\, e) \quad (11)$$

$$\text{FUNCTION EXTENSIONALITY:} \quad \frac{p \Rightarrow \mathcal{D}\, f = \mathcal{D}\, g \wedge (v \in \mathcal{D}\, f \cap \mathcal{D}\, g \Rightarrow f\, v = g\, v)}{p \Rightarrow f = g}, \quad (12)$$

with new v. Example: $\emptyset \bullet e = \emptyset \bullet e'$. Calculations are chained in the same way as with (10) but, again, the inference rule is only a bootstrap to more elegant formulations.

A consequence is abstraction equality: letting $f := v : X \mathrel{\mathbb{A}} q \;.\; d$ and $g := v : Y \mathrel{\mathbb{A}} r \;.\; e$ in $f = g \Rightarrow \mathcal{D}\, f = \mathcal{D}\, g \wedge (v \in \mathcal{D}\, f \cap \mathcal{D}\, g \Rightarrow f\, v = g\, v)$, using (4), (5) and set equality,

$$(v : X \mathrel{\mathbb{A}} q \;.\; d) = (v : Y \mathrel{\mathbb{A}} r \;.\; e) \;\Rightarrow\; (v \in X \wedge q \equiv v \in Y \wedge r) \wedge (v \in X \wedge q \Rightarrow d = e).$$

We omitted $v \in Y \wedge r$ to save space at the cost of symmetry. Extensionality (12) yields a similar metatheorem, which can be reformulated elegantly by splitting into a *domain part*, obtained by assuming $d = e$, and a *mapping part*, by assuming $Y, r = X, q$.

$$p \Rightarrow (v \in X \wedge q \equiv v \in Y \wedge r) \;\vdash\; p \Rightarrow (v : X \mathrel{\mathbb{A}} q \;.\; e) = (v : Y \mathrel{\mathbb{A}} r \;.\; e)$$
$$p \Rightarrow v \in X \wedge q \Rightarrow d = e \;\vdash\; p \Rightarrow (v : X \mathrel{\mathbb{A}} q \;.\; d) = (v : X \mathrel{\mathbb{A}} q \;.\; e)$$

with $v \notin \varphi p$. As usual, $r \vdash s$ stands for "from r one can deduce s". These rules are often tacitly invoked when calculating with abstractions.

2.3.3.3 *Operators on Functions: Generic Functionals.* Operators on functions common in mathematics impose certain restrictions on the argument functions. For instance, the usual definition of $f \circ g$ requires $\mathcal{R}\, g \subseteq \mathcal{D}\, f$ and then specifies $\mathcal{D}(f \circ g) = \mathcal{D}\, g$ and $(f \circ g)x = f\,(g\, x)$. We make such operators *generic* [Boute 2003] by removing the restrictions. The key is defining the domain of the result function to avoid out-of-domain applications in the image definition; a bonus is the design of novel operators.

Here we present only the generic functionals used in our predicate calculus; others are defined later and in [Boute 2003]. The first example defines $f \circ g$ for *any* functions f, g

$$f \circ g \;=\; x : \mathcal{D}\, g \;\mathbin{\&}\; g\; x \in \mathcal{D}\, f \;.\; f\,(g\, x). \tag{13}$$

By (4), $x \in \mathcal{D}(f \circ g) \equiv x \in \mathcal{D}\, g \wedge g\, x \in \mathcal{D}\, f$, hence $\mathcal{D}(f \circ g) = \{x : \mathcal{D}\, g \mid g\, x \in \mathcal{D}\, f\}$.

Direct extension extends operators over a set X to X-valued functions. Extension can be *monadic* ($\overline{=}$) for one-argument g, or *dyadic* ($\widehat{=}$) for 2-argument $— \star —$:

$$\overline{g}\, f \;=\; x : \mathcal{D}\, f \;\mathbin{\&}\; f\; x \in \mathcal{D}\, g \;.\; g\,(f\, x). \text{ Clearly, } \overline{g}\, f = g \circ f. \tag{14}$$

$$f\, \widehat{\star}\, g \;=\; x : \mathcal{D}\, f \cap \mathcal{D}\, g \;\mathbin{\&}\; (f\, x, g\, x) \in \mathcal{D}(\star) \;.\; (f\, x) \star (g\, x) \tag{15}$$

Variants of $\widehat{=}$ are *left* ($\overleftarrow{=}$) and *right* ($\overrightarrow{=}$) *half direct extension*:

$$f\, \overleftarrow{\star}\, e = f\, \widehat{\star}\, (\mathcal{D}\, f \bullet e) \quad \text{and} \quad e\, \overrightarrow{\star}\, f = (\mathcal{D}\, f \bullet e)\, \widehat{\star}\, f \tag{16}$$

For easy reference, we recall the constant function definer ($\bullet$) and its particularizations.

$$X \bullet e = x : X \;.\; e \text{ (provided } x \notin \varphi\, e) \qquad \varepsilon = \emptyset \bullet e \qquad d \mapsto e = \iota\, d \bullet e \tag{17}$$

Function merge ($\cup$) unites domains as far as possible. Note: $p\,?\,e' \mid e = (e, e')\, p$.

$$f \cup g = x : \mathcal{D}\, f \cup \mathcal{D}\, g \;\mathbin{\&}\; (x \in \mathcal{D}\, f \cap \mathcal{D}\, g \Rightarrow f\, x = g\, x) \;.\; (x \in \mathcal{D}\, f)\,?\, f\, x \mid g\, x \tag{18}$$

Filtering ($\downarrow$) introduces/eliminates arguments and generalizes $f = x : \mathcal{D}\, f \;.\; f\, x$. Particularization to $P := X \bullet 1$ yields the familiar *function restriction* ($\restriction$).

$$f \downarrow P \;=\; x : \mathcal{D}\, f \cap \mathcal{D}\, P \;\mathbin{\&}\; P\; x \;.\; f\, x. \tag{19}$$

$$f \restriction X \;=\; f \downarrow (X \bullet 1) \tag{20}$$

Here (and in the sequel), P is a *predicate*, that is, $\mathbb{B}$-valued function. We also extend $\downarrow$ to sets, defining $x \in (X \downarrow P) \equiv x \in X \cap \mathcal{D}\, P \wedge P\, x$. Moreover, writing d_e for $d \downarrow e$ and using partial application, this yields useful shorthands like $f_{<n}$ and $\mathbb{Z}_{>0}$ that are clear without comment, yet are entirely formal and inherit all calculation rules from $\downarrow$.

Typical relational generic functionals are *equality* and *compatibility* ($©$)

$$f \mathbin{©} g \equiv f \restriction \mathcal{D}\, g = g \restriction \mathcal{D}\, f. \tag{21}$$

For many other generic functionals and their elastic extensions, we refer to [Boute 2003]. Later on, we present some interesting ramifications for programming language design.

A very important use of generic functionals is supporting the *point-free* style, that is, without referring to domain points. The elegant algebraic flavor is illustrated next.

3. FUNCTIONAL PREDICATE CALCULUS

3.1 Introductory Remarks

This predicate calculus is called *functional* because predicates and quantifiers are functions and all calculation rules stem from function equality.

Most rules are equational (equivalences). Deriving the first few rules requires separating $\equiv$ into $\Rightarrow$ and $\Leftarrow$, but the need to do so will diminish as laws accumulate, and vanishes by the time we reach applications. Since derivation examples are perhaps less interesting to the programming community than application examples, we give only a few of them, chosen for various reasons: being typical, or just atypical, or because someone asked some question about them. Omitted derivations can be found in [Boute 2004].

3.2 Axioms and Basic Calculation Rules

3.2.1 *Predicates and Quantifiers.* A *predicate* is a function P satisfying $x \in \mathcal{D}\,P \Rightarrow P\,x \in \mathbb{B}$. We define the *quantifiers* $\forall$ and $\exists$ as predicates over predicates by the following axioms. For any predicate P,

$$\text{AXIOMS FOR QUANTIFIERS.} \quad \forall\,P \equiv P = \mathcal{D}\,P \bullet 1 \quad \text{and} \quad \exists\,P \equiv P \neq \mathcal{D}\,P \bullet 0. \tag{22}$$

We read $\forall\,P$ as "everywhere P" and $\exists\,P$ as "somewhere P". The simplicity of (22) makes all calculation rules intuitively obvious to any mathematician or engineer.

The point-free style is chosen for clarity, but familiar forms are obtained just by taking $P := x : X \;.\; p$. Then, we read $\forall\,x : X \;.\; p$ as "all x in X satisfy p" and $\exists\,x : X \;.\; p$ as "some x in X satisfy p" or any of the other commonly used phrases. The attention to the domains yields some rules that are not common in untyped variants.

Taking $P := p, q$, we can show $\forall\,(p, q) \equiv p \wedge q$; hence, $\forall$ is an elastic extension of $\wedge$.

3.2.2 *Introductory Application to Function Equality and Function Types*

3.2.2.1 *Calculation Example.* Compressing function equality (11, 12) into one equation.

$$\text{THEOREM, FUNCTION EQUALITY.} \quad f = g \equiv \mathcal{D}\,f = \mathcal{D}\,g \wedge \forall\,(f \mathrel{\widehat{=}} g). \tag{23}$$

PROOF. We show ($\Rightarrow$), the converse ($\Leftarrow$) is similar.

$$\begin{array}{ll}
f = g \Rightarrow \langle \text{Leibniz (11)} \rangle & \mathcal{D}\,f = \mathcal{D}\,g \wedge (x \in \mathcal{D}\,f \cap \mathcal{D}\,g \Rightarrow f\,x = g\,x) \\
\quad \equiv \langle p \equiv p = 1 \rangle & \mathcal{D}\,f = \mathcal{D}\,g \wedge (x \in \mathcal{D}\,f \cap \mathcal{D}\,g \Rightarrow (f\,x = g\,x) = 1) \\
\quad \equiv \langle \text{Def. } \widehat{} \text{ (15)} \rangle & \mathcal{D}\,f = \mathcal{D}\,g \wedge (x \in \mathcal{D}(f \mathrel{\widehat{=}} g) \Rightarrow (f \mathrel{\widehat{=}} g)\,x = 1) \\
\quad \equiv \langle \text{Def. } \bullet \text{ (17)} \rangle & \mathcal{D}\,f = \mathcal{D}\,g \wedge (x \in \mathcal{D}(f \mathrel{\widehat{=}} g) \Rightarrow (f \mathrel{\widehat{=}} g)x = (\mathcal{D}(f \mathrel{\widehat{=}} g) \bullet 1)x) \\
\quad \Rightarrow \langle \text{Extns. (12)} \rangle & \mathcal{D}\,f = \mathcal{D}\,g \wedge (f \mathrel{\widehat{=}} g) = \mathcal{D}(f \mathrel{\widehat{=}} g) \bullet 1 \\
\quad \equiv \langle \text{Def. } \forall \text{ (22)} \rangle & \mathcal{D}\,f = \mathcal{D}\,g \wedge \forall\,(f \mathrel{\widehat{=}} g)
\end{array}$$

Step $\langle$Extns. (12)$\rangle$ tacitly uses $\mathcal{D}\,h = \mathcal{D}\,h \cap \mathcal{D}(\mathcal{D}\,h \bullet 1)$. *Note.* $f \mathbin{©} g \equiv \forall\,(f \mathrel{\widehat{=}} g)$. □

3.2.2.2 *Function Types.* Our function concept has no (unique) *codomain* associated with it. Yet, often it is useful to indicate a set as a first approximation to or a restriction on the images. For this purpose, two familiar operators for expressing *function types* (i.e., sets of functions) are formalized here; more refined alternatives are shown later.

$$\text{The } \textit{function arrow } (\to) \text{ with } f \in X \to Y \equiv \mathcal{D}\, f = X \wedge \forall x : \mathcal{D}\, f \;.\; f\,x \in Y \quad (24)$$

$$\text{The } \textit{partial arrow } (\not\to) \text{ with } f \in X \not\to Y \equiv \mathcal{D}\, f \subseteq X \wedge \forall x : \mathcal{D}\, f \;.\; f\,x \in Y \quad (25)$$

Example. We rewrite an earlier definition as **def** $\sqrt{\ } : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ **with** $\sqrt{x}^2 = x$.

Functions of type $X \not\to Y$ are often called "partial", which is a misnomer [Boute 2000]: they are proper functions, the type just leaves the domain more loosely specified. In fact, $X \not\to Y = \bigcup (S : \mathcal{P}\, X \;.\; S \to Y)$ and, for finite X and Y with $n := |X|$ and $m := |Y|$, also $|X \to Y| = m^n$ and $|X \not\to Y| = \sum (k : 0\,..\,n \;.\; \binom{n}{k} \cdot m^k) = (m+1)^n = |X \to (Y \cup \iota\, \bot)|$.

3.2.3 *A First Collection of Calculation Rules for Quantifiers.* The axioms yield some elementary properties directly by "head calculation", such as

—for constant predicates: $\forall\,(X \bullet 1) \equiv 1$ and $\exists\,(X \bullet 0) \equiv 0$ (Proof. (17) and (22));
—for the empty predicate: $\forall\,\varepsilon \equiv 1$ and $\exists\,\varepsilon \equiv 0$ (Proof. Using $\varepsilon = \emptyset \bullet 1 = \emptyset \bullet 0$).

3.2.3.1 *Duality.* Illustrative of the algebraic style is the following theorem.

$$\text{Theorem, Duality (Generalized De Morgan).} \quad \forall\,(\overline{\neg} P) \equiv (\overline{\neg} \exists)\, P \quad (26)$$

Proof:

$$\begin{array}{lll}
\forall\,(\overline{\neg} P) & \equiv \langle \text{Def. } \forall \text{ (22), Lemma A} \rangle & \overline{\neg} P = \mathcal{D}\, P \bullet 1 \\
& \equiv \langle \text{Lemma B} \rangle & P = \overline{\neg}\,(\mathcal{D}\, P \bullet 1) \\
& \equiv \langle \text{Lemma C, } 1 \in \mathcal{D} \neg \rangle & P = \mathcal{D}\, P \bullet (\neg 1) \\
& \equiv \langle \neg 1 = 0, \text{ definition } \exists \text{ (22)} \rangle & \neg\,(\exists\, P) \\
& \equiv \langle \text{Defin. } {}^- \text{ and } \exists\, P \in \mathcal{D} \neg \rangle & \overline{\neg}\, \exists\, P
\end{array}$$ □

The following lemmata were invoked; their proofs are simple exercises [Boute 2004].

$$\text{Lemma A} : \mathcal{D}\,(\overline{\neg} P) = \mathcal{D}\, P$$
$$\text{Lemma B} : \overline{\neg} P = Q \equiv P = \overline{\neg} Q$$
$$\text{Lemma C} : x \in \mathcal{D}\, g \Rightarrow \overline{g}(X \bullet x) = g \circ (X \bullet x) = X \bullet (g\, x)$$

3.2.3.2 *Distributivity Rules.* A first batch are the following rules:

$$\begin{array}{lll}
\textit{Collecting } \forall/\wedge : & \forall\, P \wedge \forall\, Q \Rightarrow \forall\,(P \mathbin{\widehat{\wedge}} Q) & \\
\textit{Splitting } \forall/\wedge : & \mathcal{D}\, P = \mathcal{D}\, Q \Rightarrow \forall\,(P \mathbin{\widehat{\wedge}} Q) \Rightarrow \forall\, P \wedge \forall\, Q & \\
\textit{Distributivity } \forall/\wedge : & \mathcal{D}\, P = \mathcal{D}\, Q \Rightarrow (\forall\,(P \mathbin{\widehat{\wedge}} Q) \equiv \forall\, P \wedge \forall\, Q) & \text{Duals for } \exists: \\
\textit{Collecting } \exists/\vee : & \mathcal{D}\, P = \mathcal{D}\, Q \Rightarrow \exists\, P \vee \exists\, Q \Rightarrow \exists\,(P \mathbin{\widehat{\vee}} Q) & \\
\textit{Splitting } \exists/\vee : & \exists\,(P \mathbin{\widehat{\vee}} Q) \Rightarrow \exists\, P \vee \exists\, Q & \\
\textit{Distributivity } \exists/\vee : & \mathcal{D}\, P = \mathcal{D}\, Q \Rightarrow (\exists\,(P \mathbin{\widehat{\vee}} Q) \equiv \exists\, P \vee \exists\, Q) &
\end{array}$$

Merge rules for $\forall$: $\forall P \wedge \forall Q \Rightarrow \forall(P \cup Q)$
$P \mathbin{©} Q \Rightarrow \forall(P \cup Q) \Rightarrow \forall P \wedge \forall Q$
$P \mathbin{©} Q \Rightarrow (\forall(P \cup Q) \equiv \forall P \wedge \forall Q)$ (and duals for $\exists$).

PROOF (given here only for the first rule of the list)

$$\begin{array}{rcll}
\forall P \wedge \forall Q & \equiv & \langle \text{Defin. } \forall \rangle & P = \mathcal{D}\,P \bullet 1 \wedge Q = \mathcal{D}\,Q \bullet 1 \\
& \Rightarrow & \langle \text{Leibniz} \rangle & \forall(P \mathbin{\widehat{\wedge}} Q) \equiv \forall(\mathcal{D}\,P \bullet 1 \mathbin{\widehat{\wedge}} \mathcal{D}\,Q \bullet 1) \\
& \equiv & \langle \text{Defin.} \mathbin{\widehat{\ }} \rangle & \forall(P \mathbin{\widehat{\wedge}} Q) \equiv \forall x : \mathcal{D}\,P \cap \mathcal{D}\,Q \,.\, (\mathcal{D}\,P \bullet 1)x \wedge (\mathcal{D}\,Q \bullet 1)x \\
& \equiv & \langle \text{Defin. } \bullet \rangle & \forall(P \mathbin{\widehat{\wedge}} Q) \equiv \forall x : \mathcal{D}\,P \cap \mathcal{D}\,Q \,.\, 1 \wedge 1 \\
& \equiv & \langle \forall(X \bullet 1) \rangle & \forall(P \mathbin{\widehat{\wedge}} Q) \equiv 1. \qquad \square
\end{array}$$

3.2.3.3 *Rules for Equal Predicates and Monotonicity Rules.* The main ones are

$$\begin{array}{ll}
\textit{Equal predicates under } \forall : & \mathcal{D}\,P = \mathcal{D}\,Q \;\Rightarrow \forall(P \mathbin{\widehat{\equiv}} Q) \Rightarrow (\forall P \equiv \forall Q) \\
\textit{Equal predicates under } \exists : & \mathcal{D}\,P = \mathcal{D}\,Q \;\Rightarrow \forall(P \mathbin{\widehat{\equiv}} Q) \Rightarrow (\exists P \equiv \exists Q) \\
\textit{Monotonicity of } \forall/\Rightarrow : & \mathcal{D}\,Q \subseteq \mathcal{D}\,P \;\Rightarrow \forall(P \mathbin{\widehat{\Rightarrow}} Q) \Rightarrow (\forall P \Rightarrow \forall Q) \\
\textit{Monotonicity of } \exists/\Rightarrow : & \mathcal{D}\,P \subseteq \mathcal{D}\,Q \;\Rightarrow \forall(P \mathbin{\widehat{\Rightarrow}} Q) \Rightarrow (\exists P \Rightarrow \exists Q).
\end{array}$$

The latter two help chaining proof steps: $\forall(P \mathbin{\widehat{\Rightarrow}} Q)$ justifies $\forall P \Rightarrow \forall Q$ or $\exists P \Rightarrow \exists Q$ if the stated set inclusion for the domains holds.

3.2.3.4 *Quantifying Constant Predicates.* These rules show how domain issues naturally emerge without having to be continuously on guard: the rules themselves are robust.

THEOREM, CONSTANT PREDICATE UNDER $\forall$. $\forall(X \bullet p) \equiv X = \emptyset \vee p$ (27)

PROOF (with every detail shown, but optional parentheses omitted)

$$\begin{array}{rcll}
\forall(X \bullet p) & \equiv & \langle \text{Def. } \forall \text{ and } \bullet \rangle & X \bullet p = X \bullet 1 \\
& \Rightarrow & \langle \text{Leibniz (11)} \rangle & x \in X \Rightarrow (X \bullet p)x = (X \bullet 1)x \\
& \equiv & \langle \text{Definition } \bullet \rangle & x \in X \Rightarrow p = 1 \\
& \equiv & \langle \text{Prop. calc.} \rangle & (x \in X \equiv 0) \vee p \\
& \equiv & \langle x \in \emptyset \equiv 0 \rangle & (x \in X \equiv x \in \emptyset) \vee p \\
& \Rightarrow & \langle \text{Set ext. (10)} \rangle & X = \emptyset \vee p \\
X = \emptyset \vee p & \equiv & \langle p = 1 \equiv p \rangle & X = \emptyset \vee p = 1 \\
& \Rightarrow & \langle \text{Leibniz (7)} \rangle & (X \bullet p = X \bullet 1 \equiv \emptyset \bullet p = \emptyset \bullet 1) \vee p = 1 \\
& \Rightarrow & \langle \text{Leibniz (7)} \rangle & (X \bullet p = X \bullet 1 \equiv \emptyset \bullet p = \emptyset \bullet 1) \vee X \bullet p = X \bullet 1 \\
& \equiv & \langle \emptyset \bullet p = \emptyset \bullet x \rangle & X \bullet p = X \bullet 1 \vee X \bullet p = X \bullet 1 \\
& \equiv & \langle \text{Idempot. } \vee \rangle & X \bullet p = X \bullet 1 \\
& \equiv & \langle \text{Def. } \forall \text{ (22)} \rangle & \forall(X \bullet p) \qquad \square
\end{array}$$

Particular instances are $\forall \varepsilon \equiv 1$ and $\forall(X \bullet 1) \equiv 1$ and $\forall(X \bullet 0) \equiv X = \emptyset$.

The dual form $\exists(X \bullet p) \equiv X \neq \emptyset \wedge p$ is most easily obtained via duality (26) and particular instances are $\exists \varepsilon \equiv 0$ and $\exists(X \bullet 0) \equiv 0$ and $\exists(X \bullet 1) \equiv X \neq \emptyset$.

3.3 Additional Derived Proof Techniques and Rules

3.3.1 *Case Analysis, Generalized Shannon Expansion and Distributivity Rules.* Let us extend the power of case analysis and Shannon expansion to predicate calculus.

A technicality: we assign domains to free variables in expressions similarly to [Gries and Schneider 1993], letting $\mathsf{D}(v, e)$ denote the intersection of the function domains corresponding to the argument positions where v occurs free in e (defined inductively, details omitted here).

LEMMA, PARTICULARIZATION. $v \in \mathsf{D}(v, P) \wedge e \in \mathsf{D}(v, P) \Rightarrow \forall P[^v_e \Rightarrow v = e \Rightarrow \forall P$.

THEOREM, CASE ANALYSIS. $v \in \mathbb{B} \;\Rightarrow \mathbb{B} \subseteq \mathsf{D}(v, P) \Rightarrow \forall P[^v_0 \wedge \forall P[^v_1 \Rightarrow \forall P$. (28)

THEOREM, SHANNON EXPANSION. If $v \in \mathbb{B}$ and $\mathbb{B} \subseteq \mathsf{D}(v, P)$ then

$\forall P \equiv \big(v \wedge \forall P[^v_1\big) \vee \big(\neg v \wedge \forall P[^v_0\big)$ *and* $\forall P \equiv \big(\neg v \vee \forall P[^v_1\big) \wedge \big(v \vee \forall P[^v_0\big)$ (29)

$\forall P \equiv \big(v \Rightarrow \forall P[^v_1\big) \wedge \big(\neg v \Rightarrow \forall P[^v_0\big)$ (*and some evident other variants*).

3.3.1.1 *More Distributivity Rules.* Theorem (29) allows elegant proofs for the following rules, presented in point-free and pointwise form, (the latter assuming v not free in p).

Distributivity $\vee/\forall$:	$\forall(p \;\vec{\vee}\; P) \equiv p \vee \forall P$
L(eft)-distrib. $\Rightarrow/\forall$:	$\forall(p \;\vec{\Rightarrow}\; P) \equiv p \Rightarrow \forall P$
R(ight)-distr. $\Rightarrow/\exists$:	$\forall(P \;\overleftarrow{\Rightarrow}\; p) \equiv \exists P \Rightarrow p$
P(seudo)-dist. $\wedge/\forall$:	$\forall(p \;\vec{\wedge}\; P) \equiv (p \wedge \forall P) \vee \mathcal{D}\,P = \emptyset$
Distributivity $\vee/\forall$:	$\forall(v : X \;.\; p \vee q) \equiv p \vee \forall v : X \;.\; q$
L(eft)-distrib. $\Rightarrow/\forall$:	$\forall(v : X \;.\; p \Rightarrow q) \equiv p \Rightarrow \forall v : X \;.\; q$
R(ight)-distr. $\Rightarrow/\exists$:	$\forall(v : X \;.\; q \Rightarrow p) \equiv \exists(v : X \;.\; q) \Rightarrow p$
P(seudo)-dist. $\wedge/\forall$:	$\forall(v : X \;.\; p \wedge q) \equiv (p \wedge \forall v : X \;.\; q) \vee S = \emptyset$.

The names mark generalizations of rules such as $p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$ etc., while $p \wedge (q \wedge r) \equiv (p \wedge q) \wedge (p \wedge r)$ really is AC and idempotency of $\wedge$. One proof:

$$\begin{array}{lll}
\forall(p \;\vec{\vee}\; P) & \equiv \langle \text{Shannon (29)} \rangle & (p \vee \forall(0 \;\vec{\vee}\; P)) \wedge (\neg\, p \vee \forall(1 \;\vec{\vee}\; P)) \\
& \equiv \langle \text{Def. } {}^{\rightarrow}, \text{ def. } \bullet \rangle & (p \vee \forall x : \mathcal{D}\,P \;.\; 0 \vee P\,x) \wedge (\neg\, p \vee \forall x : \mathcal{D}\,P \;.\; 1 \vee P\,x) \\
& \equiv \langle \text{Prop. calc., (27)} \rangle & p \vee \forall x : \mathcal{D}\,P \;.\; P\,x \\
& \equiv \langle f = x : \mathcal{D}\,f \;.\; f\,x \rangle & p \vee \forall P.
\end{array}$$

Similar rules for $\exists$ are *distributivity* $\wedge/\exists$, *pseudodistributivity* $\vee/\exists$, *left pseudodistributivity* $\Rightarrow/\exists$ and *right pseudodistributivity* $\Rightarrow/\forall$. They are easily obtained by duality.

3.3.2 *Instantiation, Generalization and Their Use in Proving Equational Laws*

3.3.2.1 *Instantiation and Generalization.* The following theorem replace homonymous axioms in traditional logic [Mendelson 1987]. It is proven from (22) using (11, 12).

THEOREM, INSTANTIATION. $\forall P \Rightarrow e \in \mathcal{D}\,P \Rightarrow P\,e$ (30)

GENERALIZATION. $p \;\Rightarrow v \in \mathcal{D}\,P \;\Rightarrow P\,v \vdash\; p \Rightarrow \forall P$ (*fresh* v). (31)

Two typical proof techniques are captured by the following metatheorems.

METATHEOREM, $\forall$-INTRODUCTION/REMOVAL. *Assuming fresh* v,

$p \Rightarrow \forall P$ *is a theorem iff* $p \Rightarrow v \in \mathcal{D}\,P \Rightarrow P\,v$ *is a theorem.* (32)

METATHEOREM, WITNESS. *Assuming fresh* v,

$$\exists\, P \Rightarrow p \;\; \textit{is a theorem iff} \;\; v \in \mathcal{D}\, P \Rightarrow P\, v \Rightarrow p \;\; \textit{is a theorem.} \tag{33}$$

We explain their significance: for $p = 1$, (32) reflects typical implicit use of generalization: to prove $\forall\, P$, prove $v \in \mathcal{D}\, P \Rightarrow P\, v$, or assume $v \in \mathcal{D}\, P$ and prove $P\, v$.

Likewise, (33) formalizes a well-known informal proof scheme: to prove $\exists\, P \Rightarrow p$, "take" a v in $\mathcal{D}\, P$ satisfying $P\, v$ (the "witness") and prove p.

As expected, and with the usual warning, we allow weaving (31) into a calculation chain in the following way, called *generalization of the consequent*: for fresh v,

$$\begin{array}{lll} p & \Rightarrow & \langle \text{Calculation yielding } v \in \mathcal{D}\, P \Rightarrow P\, v \rangle \;\; v \in \mathcal{D}\, P \Rightarrow P\, v \\ & \Rightarrow & \langle \text{Generalization of the consequent} \rangle \;\; \forall\, P. \end{array}$$

This is used to derive a few more basic rules, but rarely (if ever) beyond.

3.3.2.2 *Trading.* The scheme just shown is illustrated in the proof of the following theorem.

THEOREM, TRADING UNDER $\forall$. $\forall\, P_Q \equiv \forall\, (Q \mathrel{\widehat{\Rightarrow}} P)$ (recall: $f_P = f \downarrow P$) (34)

PROOF. We prove only $(\Rightarrow)$; $(\Leftarrow)$ is similar. Some of the parentheses are for emphasis.

$$\begin{array}{lll} \forall\, P_Q & \Rightarrow & \langle \text{Instantiation (30)} \rangle \;\; x \in \mathcal{D}(P_Q) \Rightarrow P_Q\, x \\ & \equiv & \langle \text{Definition } \downarrow \text{ (19)} \rangle \;\; (x \in \mathcal{D}\, P \cap \mathcal{D}\, Q \wedge Q\, x) \Rightarrow P\, x \\ & \equiv & \langle \text{Shunting } \wedge \text{ to } \Rightarrow \rangle \;\; x \in \mathcal{D}\, P \cap \mathcal{D}\, Q \Rightarrow (Q\, x \Rightarrow P\, x) \\ & \equiv & \langle \text{Defin.}\widehat{\ }, \text{ remark} \rangle \;\; x \in \mathcal{D}(Q \mathrel{\widehat{\Rightarrow}} P) \Rightarrow (Q \mathrel{\widehat{\Rightarrow}} P)\, x \\ & \Rightarrow & \langle \text{Gen. consequent} \rangle \;\; \forall\, (Q \mathrel{\widehat{\Rightarrow}} P) \end{array}$$

The remark in question is $x \in \mathcal{D}\, P \cap \mathcal{D}\, Q \Rightarrow (Q\, x, P\, x) \in \mathcal{D}\, (\Rightarrow)$, by definition of predicates and $\Rightarrow$. □

From (34) and using duality (26), one can prove the $\exists$-counterpart.

THEOREM, TRADING UNDER $\exists$. $\exists\, P_Q \equiv \exists\, (Q \mathrel{\widehat{\wedge}} P)$ (35)

3.4 Expanding the Toolkit of Calculation Rules

Building a full toolkit is beyond the scope of this article and fits better in a textbook. Therefore, we just complement the preceding section with some guidelines and observations the reader will find sufficient for expanding the toolkit as needed.

3.4.1 *Some Chosen Rules for* $\forall$. For the point-free formulation, we use the following legend: let P and Q be *predicates*, R a family of predicates (i.e., $R\, x$ is a predicate for any x in $\mathcal{D}\, R$), and S a relation. The *currying* operator $—^{\mathsf{C}}$ maps $f : X \times Y \to Z$ into a f^{C} of type $X \to (Y \to Z)$ defined by $f^{\mathsf{C}}\, x\, y = f\, (x, y)$. For the *transposition* operator $—^{\mathsf{T}}$ in its simplest form, $(x : X \;.\; y : Y \;.\; e)^{\mathsf{T}} = y : Y \;.\; x : X \;.\; e$. The *range* operator $\mathcal{R}$ is defined by $y \in \mathcal{R}\, f \equiv \exists\, x : \mathcal{D}\, f \;.\; f\, x = y$. Here are some rules in point-free and pointwise form.

a. *Merge rule* $P \mathbin{©} Q \Rightarrow \forall(P \cup Q) = \forall P \wedge \forall Q$ with $\cup$ and © as in (18, 21)
b. *Transposition* $\forall(\forall \circ R) = \forall(\forall \circ R^{\mathsf{T}})$
c. *Nesting* $\forall S = \forall(\forall \circ S^{\mathsf{C}})$
d. *Composition rule* $\forall P \equiv \forall(P \circ f)$ provided $\mathcal{D}\,P \subseteq \mathcal{R}\,f$ (otherwise only $\Rightarrow$)
e. *One-point rule* $\forall P_{=e} \equiv e \in \mathcal{D}\,P \Rightarrow P\,e$

a. *Domain split* $\forall(x : X \cup Y \;.\; p) \equiv \forall(x : X \;.\; p) \wedge \forall(x : Y \;.\; p)$
b. *Dummy swap* $\forall(x : X \;.\; \forall\, y : Y \;.\; p) \equiv \forall(y : Y \;.\; \forall x : X \;.\; p)$
c. *Nesting* $\forall(x, y : X \times Y \;.\; p) \equiv \forall(x : X \;.\; \forall\, y : Y \;.\; p)$
d. *Dummy change* $\forall(y : \mathcal{R}\,f \;.\; p) \equiv \forall(x : \mathcal{D}\,f \;.\; p[^{y}_{f\,x}])$
e. *One-point rule* $\forall(x : X \wedge x = e \;.\; p) \equiv e \in X \Rightarrow p[^{x}_{e}]$

The *one-point rule* deserves a comment, since it is underrated in theory, yet found crucial in applications. Instantiation ($\forall P \Rightarrow e \in \mathcal{D}\,P \Rightarrow P\,e$) has the same right-hand side, but the one-point rule is an equivalence, hence stronger [Gries and Schneider 1993]. Its proof is also interesting:

$$\begin{array}{lll}
\forall P_{=e} \equiv & \langle\text{Filter, trading}\rangle & \forall x : X \;.\; x = e \Rightarrow P\,x \\
\Rightarrow & \langle\text{Instantiation}\rangle & e \in X \Rightarrow e = e \Rightarrow P\,e \\
\equiv & \langle\text{Reflexivity} =\rangle & e \in X \Rightarrow P\,e \text{ whereas, for the converse,}
\end{array}$$

$$\begin{array}{lll}
(e \in X \Rightarrow P\,e) \Rightarrow \forall P_{=e} & & \\
\equiv & \langle\text{Filter, trading}\rangle & (e \in X \Rightarrow P\,e) \Rightarrow \forall x : X \;.\; x = e \Rightarrow P\,x \\
\equiv & \langle\text{L-dstr.}\Rightarrow/\forall\rangle & \forall x : X \;.\; (e \in X \Rightarrow P\,e) \Rightarrow x = e \Rightarrow P\,x \\
\equiv & \langle\text{Shunting}\rangle & \forall x : X \;.\; x = e \Rightarrow (e \in X \Rightarrow P\,e) \Rightarrow P\,x \\
\equiv & \langle p \equiv 1 \Rightarrow p\rangle & \forall x : X \;.\; x = e \Rightarrow (e \in X \Rightarrow P\,e) \Rightarrow (x \in X \Rightarrow P\,x) \\
\equiv & \langle\text{Leibniz (7)}\rangle & 1.
\end{array}$$

Duals are $\exists P_{=e} \equiv e \in \mathcal{D}\,P \wedge P\,e$ and $\exists(x : X \;.\; x = e \wedge p) \equiv e \in X \wedge p[^{x}_{e}]$.

Calculating what happens when "$\Rightarrow$" in $\forall(x : \mathcal{D}\,P \;.\; x = e \Rightarrow P\,x)$ is reversed yields an interesting variant, *half-pint rule*: $\forall(x : \mathcal{D}\,P \;.\; P\,x \Rightarrow x = e) \Rightarrow \exists\, P \Rightarrow P\,e$.

3.4.2 *Rules for Swapping Quantifiers and Function Comprehension.* Dummy swap $\forall(x : X \;.\; \forall\, y : Y \;.\; p) \equiv \forall(y : Y \;.\; \forall x : X \;.\; p)$ and its $\exists$-dual support "homogeneous" swapping. For mixed swapping in one direction (proven using $p \Rightarrow q \;\equiv\; p \wedge q \;\equiv\; q$ [Gries and Schneider 1993]),

$$\text{THEOREM, SWAP } \forall \text{ OUT.} \quad \exists(y : Y \;.\; \forall x : X \;.\; p) \;\Rightarrow\; \forall(x : X \;.\; \exists\, y : Y \;.\; p). \tag{36}$$

The converse does not hold, but the following axiom, called *function comprehension*, can be seen as a "pseudo-converse": for any relation $—R—\,: Y \times X \to \mathbb{B}$,

$$\text{AXIOM.} \quad \forall(x : X \;.\; \exists\, y : Y \;.\; y\;R\;x) \;\Rightarrow\; \exists\, f : X \to Y \;.\; \forall x : X \;.\; (f\,x)\,R\,x. \tag{37}$$

This axiom (whose converse can be proven) is crucial for implicit function definitions.

A technicality: when symbols otherwise used as metavariables appear in bindings, like f in (37), they stand for variables, not expressions (evident, but worth stating).

4. APPLICATION EXAMPLES: FIRST BATCH

Applications of predicate calculus abound; one could say that it is universal in mathematics, although often tacitly. In particular, all logics, even fuzzy and intuitionistic ones, use "common" predicate logic at the metalevel.

Here we consider only examples that illustrate the particular flavor distinguishing our approach from others. Some examples are endogenous, prompted by issues within predicate calculus or mathematics, others are exogenous, prompted by issues elsewhere.

4.1 Endogenous Application Examples

4.1.1 *The Empty Domain Issue: A Fresh View on "Andrew's Challenge".* Manolios and Moore [2001] observed that many calculational predicate logics are error-prone due to insufficient attention to types of variables and function domains. In our formalism, exactly those items form the basis of the axiomatization, thereby avoiding the errors and making calculation "robust". This is what will be illustrated here.

First, we clear up some terminology. Commonly, the "extent" of ad hoc abstractors like $\forall$ and $\sum$ is syntactically linked to the symbol (as in $\forall x{:}X.$ and $\sum_{i=m}^{n}$) and called the *range*. This notion differs from *function range*. Moreover, in Funmath, this "extent" is not associated with the abstractor but is the *domain* of the function appearing as the argument of the elastic operator (e.g., $\mathcal{D}\,P$ in $\forall\,P$ and $\mathcal{D}\,f$ in $\sum\,f$). We call this *domain modulation*, and it applies by design to all elastic operators. Hence, we decide to avoid confusion and separate nomenclature by using the term *domain*.

Andrew's challenge is the term used by Gries [1996b] for calculationally proving

$$((\exists x \forall y \mid :\ p.x \equiv p.y) \equiv ((\exists x \mid :\ q.x) \equiv (\forall y \mid :\ p.y))) \equiv$$
$$((\exists x \forall y \mid :\ q.x \equiv q.y) \equiv ((\exists x \mid :\ p.x) \equiv (\forall y \mid :\ q.y))),$$

(notation from [Gries 1996b]). The stronger $(\exists x \forall y \mid :\ p.x \equiv p.y) \equiv (\exists x \mid :\ p.x) \equiv (\forall y \mid :\ p.y)$ has been proven in different ways by Gries [1996b], Dijkstra [1996a] and independently by Rajeev Joshi and Kedar Sharad Namjoshi, as shown in [Dijkstra 1996b].

The latter is direct, but proves only what is asked, without useful "side results". The reader can easily paraphrase this proof in our formalism (see [Boute 2004]) and observe that the ad hoc shorthands $[t] \equiv \langle \forall x :: t.x \rangle$ and $\langle t \rangle \equiv \langle \exists x :: t.x \rangle$ introduced in [Dijkstra 1996b] to keep formulas sort are obviated in Funmath, writing $\forall\,P$ and $\exists\,P$ in point-free style. So Andrew's Challenge becomes: $\exists\,P \equiv \forall\,P \equiv \exists x : \mathcal{D}\,P\,.\,\forall\,y : \mathcal{D}\,P\,.\,P\,x \equiv P\,y$.

More interesting is the proof in [Dijkstra 1996a] meant to reveal a pitfall requiring case distinction between empty and nonempty "range", information that is unavailable in the quantified formulas used. In our formalism, domain modulation together with the calculation rules make it appear exactly when needed. Our proof reflects the issues in [Dijkstra 1996a], but arose in a quite

different context: given the *constant function predicate* con

$$\mathsf{con}\ f \ \equiv\ \forall x : \mathcal{D}\ f\ .\ \forall y : \mathcal{D}\ f\ .\ f\ x = f\ y, \tag{38}$$

we wanted to see to what extent $\mathsf{con}\ f$ equals $\exists x : \mathcal{D}\ f\ .\ f\ =\ \mathcal{D}\ f \bullet f\ x$ by exploring it *formally* (to be safe) and make adaptations as calculations indicate. We start by observing that $f\ =\ \mathcal{D}\ f \bullet f\ x \equiv \forall y : \mathcal{D}\ f\ .\ f\ y\ =\ f\ x$ by (23, 17) and calculate

$$\begin{array}{ll}
\exists x : \mathcal{D}\ f\ .\ \forall y : \mathcal{D}\ f\ .\ f\ y = f\ x & \\
\quad \equiv & \langle \text{Idempotency} \wedge \rangle\ \ \exists x : \mathcal{D}\ f\ .\ \forall (y : \mathcal{D}\ f\ .\ f\ y = f\ x) \wedge \forall (z : \mathcal{D}\ f\ .\ f\ z = f\ x) \\
\quad \equiv & \langle \text{Distribut.}\ \wedge/\forall \rangle\ \ \exists x : \mathcal{D}\ f\ .\ \forall (y : \mathcal{D}\ f\ .\ f\ y = f\ x \wedge \forall z : \mathcal{D}\ f\ .\ f\ z = f\ x) \\
\quad \equiv & \langle \text{Distribut.}\ \wedge/\forall \rangle\ \ \exists x : \mathcal{D}\ f\ .\ \forall y : \mathcal{D}\ f\ .\ \forall z : \mathcal{D}\ f\ .\ f\ y = f\ x \wedge f\ z = f\ x \\
\quad \Rightarrow & \langle \text{Transitivity} = \rangle\ \ \exists x : \mathcal{D}\ f\ .\ \forall y : \mathcal{D}\ f\ .\ \forall z : \mathcal{D}\ f\ .\ f\ y = f\ z \\
\quad \equiv & \langle \text{Definition con} \rangle\ \ \exists x : \mathcal{D}\ f\ .\ \mathsf{con}\ f \\
\quad \equiv & \langle \text{Constant pred.} \rangle\ \ \mathcal{D}\ f \neq \emptyset \wedge \mathsf{con}\ f.
\end{array}$$

The idempotency step may seem surprising, but is (i) a standard technique for inserting a second $\forall$, (ii) a way to enable an essential property of equality, viz., transitivity. Unfortunately, this broke the $\equiv$ chain, so the converse has to be explored separately:

$$\begin{array}{ll}
\mathcal{D}\ f \neq \emptyset \wedge \mathsf{con}\ f \ \equiv & \langle \text{Definition con} \rangle\ \mathcal{D}\ f \neq \emptyset \wedge \forall x : \mathcal{D}\ f\ .\ \forall y : \mathcal{D}\ f\ .\ f\ x = f\ y \\
\qquad \Rightarrow & \langle \text{Remark below} \rangle\ \exists x : \mathcal{D}\ f\ .\ \forall y : \mathcal{D}\ f\ .\ f\ x = f\ y, \text{ so we have:}
\end{array}$$

LEMMA D. $\mathcal{D}\ f \neq \emptyset \wedge \mathsf{con}\ f\ \equiv\ \exists x : \mathcal{D}\ f\ .\ \forall y : \mathcal{D}\ f\ .\ f\ x = f\ y.$

Remark. The first line (with $\forall$) and the goal (with $\exists$) suggests calculating $\forall P \Rightarrow \exists P$:

$$\begin{array}{lll}
\forall P \Rightarrow \exists P & \equiv \langle \text{From} \Rightarrow \text{to} \vee \rangle & \neg \forall P \vee \exists P \\
& \equiv \langle \text{Duality (26)} \rangle & \exists (\neg P) \vee \exists P \\
& \equiv \langle \text{Distrib.}\ \exists/\vee \rangle & \exists (\neg P\ \widehat{\vee}\ P) \\
& \equiv \langle \text{Excl. middle} \rangle & \exists (\mathcal{D}\ P \bullet 1) \\
& \equiv \langle \text{Const. pred.} \rangle & \mathcal{D}\ P \neq \emptyset, \text{ which proves:}
\end{array}$$

LEMMA E. $\forall P \Rightarrow \exists P \equiv \mathcal{D}\ P \neq \emptyset.$

Hence, $\mathcal{D}\ P\ \neq\ \emptyset \wedge \forall P \Rightarrow \exists P$. Taking $P := \forall y : \mathcal{D}\ f\ .\ f\ x\ =\ f\ y$ completes the remark.

Lemma D together with $(\neg\ p \wedge q \equiv r) \Rightarrow (p \vee q \equiv p \vee r)$ and $p \Rightarrow q \equiv p \vee q \equiv q$ and $\mathcal{D}\ f\ =\ \emptyset \Rightarrow \mathsf{con}\ f$, and recalling $f\ =\ \mathcal{D}\ f \bullet f\ x \equiv \forall y : \mathcal{D}\ f\ .\ f\ y\ =\ f\ x$, yields the

CONSTANT FUNCTION THEOREM. $\mathsf{con}\ f \equiv \mathcal{D}\ f = \emptyset \vee \exists x : \mathcal{D}\ f\ .\ f = \mathcal{D}\ f \bullet f\ x.$ (39)

Our proof of Andrew's Challenge arises as a by-product. Indeed, Lemma D specializes to $\mathcal{D}\ P\ \neq\ \emptyset \wedge \mathsf{con}\ P \equiv \exists x : \mathcal{D}\ P\ .\ \forall y : \mathcal{D}\ P\ .\ P\ x \equiv P\ y$. Since $\mathcal{D}\ P\ \neq\ \emptyset \equiv \forall P \Rightarrow \exists P$, only $\mathsf{con}\ P$ is left to calculate. The result clearly completes the

desired proof.

$$\begin{array}{rcll} \mathsf{con}\, P & \equiv & \langle \text{Definition con} \rangle & \forall x : \mathcal{D}\, P \,.\, \forall y : \mathcal{D}\, P \,.\, P\, x \equiv P\, y \\ & \equiv & \langle \text{Remark below} \rangle & \forall x : \mathcal{D}\, P \,.\, \forall y : \mathcal{D}\, P \,.\, P\, x \Rightarrow P\, y \\ & \equiv & \langle \text{L-distr.} \Rightarrow/\forall \rangle & \forall x : \mathcal{D}\, P \,.\, P\, x \Rightarrow \forall\, P \\ & \equiv & \langle \text{R-distr.} \Rightarrow/\exists \rangle & \exists\, P \Rightarrow \forall\, P, \text{ which proves:} \end{array}$$

LEMMA F. $\mathsf{con}\, P \equiv \exists\, P \Rightarrow \forall\, P$.

Remark. Uneventful steps not written out are: split $P\, x \equiv P\, y$ as mutual implication, use distributivity $\forall/\wedge$ to obtain conjunction of two quantified formulas, swap $\forall$'s in one and rename dummies to make these formulas the same, use idempotency of $\wedge$.

As promised, we have seen how the formal rules make the domain emptiness issue emerge exactly when relevant, without requiring to beware of it, so the formalism is "robust" against oversights. More strongly: we have seen how the shape of the formulas provides clues about how to proceed. To highlight this, we adopted the somewhat unusual format of stating lemmata and theorems after deriving them. Recasting this example into the "finished product" format as in textbooks is a simple exercise.

4.1.2 *The Function Range, with an Application to Set Comprehension*

4.1.2.1 *Function Range.* We define the *function range* operator $\mathcal{R}$ by the axiom

$$\text{AXIOM, FUNCTION RANGE.} \quad e \in \mathcal{R}\, f \equiv \exists (x : \mathcal{D}\, f \,.\, f\, x = e). \tag{40}$$

Equivalently, in point-free style: $e \in \mathcal{R}\, f \equiv \exists (f \doteq e)$. A useful familiarization exercise is proving $f \in X \to Y \equiv \mathcal{D}\, f = X \wedge \mathcal{R}\, f \subseteq Y$ given $Z \subseteq Y \equiv \forall z : Z \,.\, z \in Y$.

4.1.2.2 *A Very Useful Theorem.* This point-free theorem underlies "dummy change". The proof is particularly illustrative for the one-point rule and right distributivity $\Rightarrow/\exists$.

THEOREM, COMPOSITION RULE.

$$\text{(i) } \forall\, P \Rightarrow \forall (P \circ f) \textit{ and } \text{(ii) } \mathcal{D}\, P \subseteq \mathcal{R}\, f \Rightarrow (\forall (P \circ f) \equiv \forall\, P) \tag{41}$$

We prove the common part; items (i) and (ii) follow in one more step each.

$$\begin{array}{rcll} \forall (P \circ f) \equiv & \langle \text{Definition} \circ \rangle & \forall x : \mathcal{D}\, f \wedge f\, x \in \mathcal{D}\, P \,.\, P\, (f\, x) \\ \equiv & \langle \text{Trading sub} \forall \rangle & \forall x : \mathcal{D}\, f \,.\, f\, x \in \mathcal{D}\, P \Rightarrow P\, (f\, x) \\ \equiv & \langle \text{One-point rule} \rangle & \forall x : \mathcal{D}\, f \,.\, \forall y : \mathcal{D}\, P \,.\, y = f\, x \Rightarrow P\, y \\ \equiv & \langle \text{Swap under} \forall \rangle & \forall y : \mathcal{D}\, P \,.\, \forall x : \mathcal{D}\, f \,.\, y = f\, x \Rightarrow P\, y \\ \equiv & \langle \text{R-dstr.} \Rightarrow/\exists \rangle & \forall y : \mathcal{D}\, P \,.\, \exists (x : \mathcal{D}\, f \,.\, y = f\, x) \Rightarrow P\, y \\ \equiv & \langle \text{Definition} \mathcal{R} \rangle & \forall y : \mathcal{D}\, P \,.\, y \in \mathcal{R}\, f \Rightarrow P\, y. \end{array}$$

The dual is $\exists (P \circ f) \Rightarrow \exists\, P$ and $\mathcal{D}\, P \subseteq \mathcal{R}\, f \Rightarrow \exists (P \circ f) \equiv \exists\, P$, whereas the pointwise variant is "dummy change": $\forall (y : \mathcal{R}\, f \,.\, p) \equiv \forall (x : \mathcal{D}\, f \,.\, p[^{y}_{f\, x}])$ provided $\mathcal{R}\, f \subseteq \mathsf{D}\ (y, p)$.

4.1.2.3 *Set Comprehension.* Introducing {—} as an operator *fully interchangeable* with $\mathcal{R}$, expressions like $\{2,3,5\}$ and $Even = \{m:\mathbb{Z} \;.\; 2\cdot m\}$ have a familiar form and meaning.

Indeed, since tuples are functions, $\{e,e',e''\}$ denotes a set by listing its elements. Also, $k \in \{m:\mathbb{Z} \;.\; 2\cdot m\} \equiv \exists\, m:\mathbb{Z} \;.\; k = 2\cdot m$ by (40). To cover common forms (without their flaws), recall that abstraction has two variants $e \mid x:X$ standing for $x:X \;.\; e$ and $x:X \mid p$ for $x:X \wedge p \;.\; x$, yielding expressions like $\{2\cdot m \mid m:\mathbb{Z}\}$ and $\{m:\mathbb{N} \mid m < n\}$.

The only "custom" to be discarded is using {} for singletons. This is no loss, since respecting it would violate Leibniz's principle, for example, $f = a,b \Rightarrow \{f\} = \{a,b\}$. With our convention, this is consistent. Yet, to avoid baffling the uninitiated, we advise writing $\mathcal{R}\, f$ rather than $\{f\}$ if f is a function identifier. Hence, we keep both symbols in stock.

Now binding is always trouble-free, even in $\{n:\mathbb{Z} \mid n \in Even\} = \{n:Even \mid n \in \mathbb{Z}\}$ and $\{n \in \mathbb{Z} \mid n:Even\} \neq \{n \in Even \mid n:\mathbb{Z}\}$. All desired calculation rules follow from predicate calculus by the axiom for $\mathcal{R}$. A repetitive pattern is captured by

$$\text{THEOREM, SET COMPREHENSION.} \quad e \in \{x:X \mid p\} \equiv e \in X \wedge p[^x_e] \tag{42}$$

$$\begin{aligned} \text{PROOF.} \quad e \in \{x:X \wedge p \;.\; x\} &\equiv \langle \text{Def. range (40)} \rangle \; \exists\, x:X \wedge p \;.\; x = e \\ &\equiv \langle \text{Trading, twice} \rangle \; \exists\, x:X \wedge x = e \;.\; p \\ &\equiv \langle \text{One-point rule} \rangle \; e \in X \wedge p[^x_e] \qquad \square \end{aligned}$$

Furthermore, for the quantifiers $Q:\{\forall,\exists\}$

$$\text{THEOREM, COMPREHENSION TRADING.} \quad Q(w:\{e \mid v:V\} \;.\; p) \equiv Q\big(v:V \;.\; p[^v_e]\big). \tag{43}$$

PROOF. Let $f := e \mid v:V$ and $P := w:\mathcal{R}\, f \;.\; p$, in the composition rule.

4.1.2.4 *Calculation Examples.* The chosen topic is partial application, also relevant to (functional) programming, for example, Haskell [Hudak et al. 1999] supports a variant called *sections*. For any infix $\star$ and any x, we define the domain of $(x\star)$ by $\mathcal{D}(x\star) = \{y \mid u,y:\mathcal{D}(\star) \wedge u = x\}$.

A first example is calculating $\mathcal{R}(x\star)$, apart from $\mathcal{R}(x\star) = \{x\star y \mid y:\mathcal{D}(x\star)\}$.

$$\begin{aligned} z \in \mathcal{R}(x\star) \;\equiv\; & \langle \text{Definition } \mathcal{R} \rangle \;\; \exists\, v:\mathcal{D}(x\star) \;.\; z = x\star v \\ \equiv\; & \langle \text{Defin. } \mathcal{D}(x\star) \rangle \;\; \exists\, v:\{y \mid u,y:\mathcal{D}(\star) \wedge u = x\} \;.\; z = x\star v \\ \equiv\; & \langle \text{Compr. trad.} \rangle \;\; \exists\, u,y:\mathcal{D}(\star) \wedge u = x \;.\; z = x\star y \\ \equiv\; & \langle \text{Definition } \mathcal{R} \rangle \;\; z \in \{x\star y \mid u,y:\mathcal{D}(\star) \wedge u = x\} \end{aligned}$$

Hence, we found $\mathcal{R}(x\star) = \{x\star y \mid u,y:\mathcal{D}(\star) \wedge u = x\}$ (not surprising, but in such matters a formal proof is not a luxury). A second example is verifying the conjecture $x \,\vec{\star}\, f = (x\star)\circ f$, which turns out to hold but is left as a quite pleasant exercise.

Finally, we verify the conjecture $\mathcal{D}(\star) = \bigcup x : \{x \mid x, y : \mathcal{D}(\star)\} \,.\, \{x, y \mid y : \mathcal{D}(x \star)\}$.

$$
\begin{array}{lll}
x, y \in \bigcup x : \{x \mid x, y : \mathcal{D}(\star)\} \,.\, \{x, y \mid y : \mathcal{D}(x \star)\} & & \\
\quad \equiv & \langle \text{Definition } \bigcup \rangle & \exists u : \{x \mid x, y : \mathcal{D}(\star)\} \,.\, x, y \in \{u, y \mid y : \mathcal{D}(u \star)\} \\
\quad \equiv & \langle \text{Definition } \mathcal{R} \rangle & \exists u : \{x \mid x, y : \mathcal{D}(\star)\} \,.\, \exists v : \mathcal{D}(u \star) \,.\, x, y = u, v \\
\quad \equiv & \langle \text{Func. equal.} \rangle & \exists u : \{x \mid x, y : \mathcal{D}(\star)\} \,.\, \exists v : \mathcal{D}(u \star) \,.\, x = u \wedge y = v \\
\quad \equiv & \langle \text{One-pt. rule} \rangle & \exists u : \{x \mid x, y : \mathcal{D}(\star)\} \,.\, y \in \mathcal{D}(u \star) \wedge x = u \\
\quad \equiv & \langle \text{One-pt. rule} \rangle & x \in \{x \mid x, y : \mathcal{D}(\star)\} \wedge y \in \mathcal{D}(x \star) \\
\quad \equiv & \langle \text{Def. } \mathcal{D}(x \star) \rangle & x \in \{x \mid x, y : \mathcal{D}(\star)\} \wedge y \in \{y \mid u, y : \mathcal{D}(\star) \wedge u = x\} \\
\quad \equiv & \langle \text{Definition } \mathcal{R} \rangle & \exists (u, v : \mathcal{D}(\star) \,.\, x = u) \wedge \exists (u, v : \mathcal{D}(\star) \wedge u = x \,.\, v = y) \\
\quad \equiv & \langle \text{Trading} \backslash \exists \rangle & \exists (u, v : \mathcal{D}(\star) \,.\, x = u) \wedge \exists (u, v : \mathcal{D}(\star) \,.\, u = x \wedge v = y) \\
\quad \equiv & \langle \text{Distrib. } \exists / \wedge \rangle & \exists u, v : \mathcal{D}(\star) \,.\, x = u \wedge u = x \wedge v = y \\
\quad \equiv & \langle \text{Func. equal.} \rangle & \exists u, v : \mathcal{D}(\star) \,.\, u, v = x, y \\
\quad \equiv & \langle \text{One-pt. rule} \rangle & x, y \in \mathcal{D}(\star)
\end{array}
$$

So the conjecture holds. *Note*: Previous exposure to sloppy practice may cause people to read $x, y \in X$ wrongly as $x \in X \wedge y \in X$. When in doubt, parenthesize all pairs.

Additional rules shorten the proof but presenting them would lengthen the article. A similar exercise is verifying $\mathcal{R}(\star) = \bigcup x : \{x \mid x, y : \mathcal{D}(\star)\} \,.\, \mathcal{R}(x \star)$.

4.2 Exogenous Application Examples

We start with an example quite removed from computing, making it all the more apparent how much the later proofs in computing are similar and how predicate calculus contributes to the methodological unification of widely different disciplines.

The examples show how calculation is guided by the shape of the formulas rather than by their meaning. Some proofs may appear unintuitive to those used to the latter approach only, but the opposite is true: they mark a "parallel" intuition for syntactic reasoning, complementing existing "semantic" approaches to problem solving and discovery. Like all forms of intuition, it can be acquired only by active participation.

4.2.1 *Limits in Mathematical Analysis*

4.2.1.1 *Motivation.* In a comment about the first chapter of his book Taylor [2000] observes:

> The notation of elementary school arithmetic, which nowadays everyone takes for granted, took centuries to develop. There was an intermediate stage called *syncopation*, using abbreviations for the words for addition, square, root, *etc*. For example Rafael Bombelli (*c*. 1560) would write
>
> $$\text{R. c. L. 2 p. di m. 11 L} \quad \text{for our} \quad \sqrt[3]{2 + 11i}.$$
>
> Many professional mathematicians to this day use the quantifiers (∀,∃) in a similar fashion,
>
> $$\exists \delta > 0 \text{ s.t. } |f(x) - f(x_0)| < \epsilon \text{ if } |x - x_0| < \delta, \text{ for all } \epsilon > 0,$$
>
> in spite of [...] Even now, mathematics students are expected to learn

> complicated (ϵ-δ)-proofs in analysis with no help in understanding the logical structure of the arguments. Examiners fully deserve the garbage that they get in return.

Key words in the quotation are *"understanding the logical structure of the arguments"*. This is exactly one of the advantages formal calculational predicate logic offers over the traditional informal arguments in words or using $\forall$ and $\exists$ as mere syncopation.

Even if some proofs in words are as good as this medium permits, there is a chafing style breach between calculating derivatives and integrals, where symbols really do the work, and the logical arguments supporting these rules, which are felt as a burden.

Bridging this gap is illustrated by recasting a classical introduction to limits, taken from Lang's excellent *Undergraduate Analysis* [Lang 1983], into our predicate calculus.

Although analysis may not interest all members of the programming community, we feel that at least a few examples are needed to convey the flavor and the wide scope of the formalism. The full treatment and the omitted proofs can be found in [Boute 2004].

4.2.1.2 *Proof Examples.* We first define *adherence*. Formalizing the version in Lang [1983] yields

$$\textbf{def}\ \mathsf{Ad} : \mathcal{P}\,\mathbb{R} \to \mathcal{P}\,\mathbb{R}\ \ \textbf{with}\ \mathsf{Ad}\ S = \{v : \mathbb{R} \mid \forall \epsilon : \mathbb{R}_{>0}\,.\,\exists x : S\,.\,|x - v| < \epsilon\}. \quad (44)$$

Although we have adequate calculation rules for set expressions, formulations with predicates based on the obvious isomorphism between $\mathcal{P}\,X$ and $X \to \mathbb{B}$ are shorter and make calculations more homogeneous. Therefore, we proceed from the definitions

$$\begin{array}{l}
\textbf{def}\ \mathsf{ad} : (\mathbb{R} \to \mathbb{B}) \to (\mathbb{R} \to \mathbb{B})\ \textbf{with}\ \mathsf{ad}\ P\,v \equiv \forall \epsilon : \mathbb{R}_{>0}\,.\,\exists x : \mathbb{R}_P\,.\,|x - v| < \epsilon \\
\textbf{def}\ \mathsf{open} : (\mathbb{R} \to \mathbb{B}) \to \mathbb{B}\ \ \textbf{with} \\
\quad \mathsf{open}\ P \equiv \forall v : \quad \mathbb{R}_P\,.\,\exists \epsilon : \mathbb{R}_{>0}\,.\,\forall x : \mathbb{R}\ .\,|x - v| < \epsilon \Rightarrow P\,x \\
\textbf{def}\ \mathsf{colsed} : (\mathbb{R} \to \mathbb{B}) \to \mathbb{B}\ \ \textbf{with}\ \mathsf{colsed}\ P \equiv \mathsf{open}\ (\neg P).
\end{array}$$

The latter reformulate the usual concepts of *open* and *closed sets*. We take two exercises from Lang [1983] because the proofs show striking similarities with later theorems directly relevant to programming, both in the issues raised and in the shape of the formulas. The exercises consist in proving two properties. Both use $P\,v \Rightarrow \mathsf{ad}\ P\,v$ (easy to show).

CLOSURE PROPERTY. $\mathsf{colsed}\ P \equiv \mathsf{ad}\ P = P$

PROOF

$$\begin{array}{ll}
\mathsf{colsed}\ P & \\
\equiv & \langle \text{Definit. colsed} \rangle \quad \mathsf{open}\ (\neg P) \\
\equiv & \langle \text{Definit. open} \rangle \quad \forall v : \mathbb{R} \neg_P\,.\,\exists \epsilon : \mathbb{R}_{>0}\,.\,\forall x : \mathbb{R}\ .\,|x - v| < \epsilon \Rightarrow \neg P\,x \\
\equiv & \langle \text{Trading sub } \forall \rangle \quad \forall v : \mathbb{R}\ .\,\neg P\,v \Rightarrow \exists \epsilon : \mathbb{R}_{>0}\,.\,\forall x : \mathbb{R}\ .\,|x - v| < \epsilon \Rightarrow \neg P\,x \\
\equiv & \langle \text{Contrapositive} \rangle \quad \forall v : \mathbb{R}\ .\,\neg \exists (\epsilon : \mathbb{R}_{>0}\,.\,\forall x : \mathbb{R}\ .\,P\,x \Rightarrow \neg(|x - v| < \epsilon)) \Rightarrow P\,v \\
\equiv & \langle \text{Duality, twice} \rangle \quad \forall v : \mathbb{R}\ .\,\forall (\epsilon : \mathbb{R}_{>0}\,.\,\exists x : \mathbb{R}\ .\,P\,x \wedge |x - v| < \epsilon) \Rightarrow P\,v \\
\equiv & \langle \text{Definition ad} \rangle \quad \forall v : \mathbb{R}\ .\,\mathsf{ad}\ P\,v \Rightarrow P\,v \\
\equiv & \langle P\,v \Rightarrow \mathsf{ad}\ P\,v \rangle \quad \forall v : \mathbb{R}\ .\,\mathsf{ad}\ P\,v \equiv P\,v.
\end{array}$$

IDEMPOTENCY OF ADHERENCE. $\mathsf{ad} \circ \mathsf{ad} = \mathsf{ad}$

PROOF. By function equality and the definition of $\circ$, proving $\mathsf{ad}\,(\mathsf{ad}\,P)\,v \equiv \mathsf{ad}\;P\,v$ suffices. Instantiating $P\,v \Rightarrow \mathsf{ad}\,P\,v$ yields $\mathsf{ad}\,P\,v \Rightarrow \mathsf{ad}\,(\mathsf{ad}\,P)\,v$. For the converse,

$$
\begin{array}{ll}
\mathsf{ad}\,(\mathsf{ad}\,P)\,v & \\
\equiv \langle \text{Definition ad} \rangle & \forall \epsilon : \mathbb{R}_{>0} \,.\, \exists x : \mathbb{R}_{\mathsf{ad}\;P} \,.\, |x - v| < \epsilon \\
\equiv \langle \text{Trading sub } \exists \rangle & \forall \epsilon : \mathbb{R}_{>0} \,.\, \exists x : \mathbb{R} \;.\; \mathsf{ad}\,P\,x \wedge |x - v| < \epsilon \\
\equiv \langle \text{Definition ad} \rangle & \forall \epsilon : \mathbb{R}_{>0} \,.\, \exists x : \mathbb{R} \;.\; \forall (\epsilon : \mathbb{R}_{>0} \,.\, \exists z : \mathbb{R}_P \,.\, |z - x| < \epsilon) \wedge |x - v| < \epsilon \\
\Rightarrow \langle \text{Instantiation} \rangle & \forall \epsilon : \mathbb{R}_{>0} \,.\, \exists x : \mathbb{R} \;.\; \exists (z : \mathbb{R}_P \,.\, |z - x| < \epsilon) \wedge |x - v| < \epsilon \\
\equiv \langle \text{Distrib. } \wedge/\exists \rangle & \forall \epsilon : \mathbb{R}_{>0} \,.\, \exists x : \mathbb{R} \;.\; \exists z : \mathbb{R}_P \,.\, |z - x| < \epsilon \wedge |x - v| < \epsilon \\
\equiv \langle \text{Monot. } + /< \rangle & \forall \epsilon : \mathbb{R}_{>0} \,.\, \exists x : \mathbb{R} \;.\; \exists z : \mathbb{R}_P \,.\, |z - x| + |x - v| < 2 \cdot \epsilon \\
\Rightarrow \langle \text{Triangle ineq.} \rangle & \forall \epsilon : \mathbb{R}_{>0} \,.\, \exists x : \mathbb{R} \;.\; \exists z : \mathbb{R}_P \,.\, |z - v| < 2 \cdot \epsilon \\
\equiv \langle \text{Dummy swap} \rangle & \forall \epsilon : \mathbb{R}_{>0} \,.\, \exists z : \mathbb{R}_P \,.\, \exists x : \mathbb{R} \;.\; |z - v| < 2 \cdot \epsilon \\
\equiv \langle \text{Const. pred.} \rangle & \forall \epsilon : \mathbb{R}_{>0} \,.\, \exists z : \mathbb{R}_P \,.\, |z - v| < 2 \cdot \epsilon \\
\equiv \langle \text{Dummy chng.} \rangle & \forall \epsilon : \mathbb{R}_{>0} \,.\, \exists z : \mathbb{R}_P \,.\, |z - v| < \epsilon \\
\equiv \langle \text{Definition ad} \rangle & \mathsf{ad}\,P\,v.
\end{array}
$$

Note again how the calculation rules for quantification are a major help in deciding at every step what to do next. Although brief, this example shows "how to rewrite your favorite analysis text and exercise mathematics for computing science in the process".

4.2.1.3 *Limits: Conventional Treatment.* A representative example [Lang 1983] is:

Definition. Let S be a set of numbers and a be adherent to S. Let f be a function defined on S. We shall say that the *limit of* $f(x)$ *as* x *approaches* a *exists,* if there exists a number L having the following property. Given ϵ, there exists a number $\delta > 0$ such that for all $x \in S$ satisfying $|x - a| < \delta$ we have $|f(x) - L| < \epsilon$. If that is the case, we write

$$\lim_{\substack{x \to a \\ x \in S}} f(x) = L.$$

[...] PROPOSITION 2.1. *Let S be a set of numbers, and assume that a is adherent to S. Let S' be a subset of S, and assume that a is also adherent to S'. Let f be a function defined on S. If $\lim_{x \to a, x \in S} f(x)$ exists, then $\lim_{x \to a, x \in S'} f(x)$ also exists, and those limits are equal. In particular, if the limit exists, it is unique.*

PROOF. (Proof in words, not reproduced here). □

4.2.1.4 *Discussion, Formal Treatment.* The definition reflects a custom, found in many mathematics texts, of writing an equality when the definition actually characterizes a relation; in this example: $\lim_{x \to a, x \in S} f(x) = L$. This is not innocuous abuse of notation[1]: It ruins the argument. For instance, it suggests fallacious uniqueness proofs like: let L and M satisfy $\lim_{x \to a, x \in S} f(x) = L$ and $\lim_{x \to a, x \in S} f(x) = M$, then $L = M$ by transitivity.

Some mathematicians who have grown up with this custom may argue that they have learned how to be careful, but thereby abandon all hope for safe formal

[1]Abuse of notation is never innocuous; even when temptation arises, it reveals a conceptual flaw.

reasoning. Moreover, there is no reason for perpetuating flaws if prevention is known and easy.

For this case, a proper relational formulation suffices. For this example, we define a relation islim_f (parameterized by any function $f : \mathbb{R} \nrightarrow \mathbb{R}$) between $\mathbb{R}$ and the set of points adherent to $\mathcal{D}\, f$, that is: $\mathsf{islim}_f \in \mathbb{R} \times \mathsf{Ad}\,(\mathcal{D}\, f) \to \mathbb{B}$. We shall see later how such types are expressed formally. The image definition is

$$L\ \mathsf{islim}_f\ a \ \equiv\ \forall \epsilon : \mathbb{R}_{>0}\,.\,\exists \delta : \mathbb{R}_{>0}\,.\,\forall x : \mathcal{D}\, f\ .\ |x-a| < \delta \Rightarrow |f\, x - L| < \epsilon. \quad (45)$$

The affix convention is chosen such that $L\ \mathsf{islim}_f\ a$ is read "L is a limit for f at a". Domain modulation (via $\mathcal{D}\, f$) subsumes S in the conventional formulation.

PROPOSITION 2.1, FORMALIZED. For any function $f : \mathbb{R} \nrightarrow \mathbb{R}$, any subset S of $\mathcal{D}\, f$ and any a adherent to S,

(i) $\exists (L : \mathbb{R}\ .\ L\ \mathsf{islim}_f\ a) \Rightarrow \exists (L : \mathbb{R}\ .\ L\ \mathsf{islim}_{f \restriction S}\ a)$,

(ii) $\forall L\ : \mathbb{R}\ .\ \forall M : \mathbb{R}\ .\ L\ \mathsf{islim}_f\ a \wedge M\ \mathsf{islim}_{f \restriction S}\ a \Rightarrow L = M$.

The calculational proofs are very similar to those for adherence and are given in Boute [2004].

Our calculational approach to limits is completed by defining a functional lim such that, for any argument $f : \mathbb{R} \nrightarrow \mathbb{R}$, the function $\mathsf{lim}\ f$ is defined by

$$\begin{aligned} &\mathsf{lim}\ f\ \in \{a : \mathsf{Ad}\,(\mathcal{D}\, f) \mid \exists b : \mathbb{R}\ .\ b\ \mathsf{islim}_f\ a\} \to \mathbb{R} \\ &\forall a : \mathcal{D}(\mathsf{lim}\ f)\ .\ (\mathsf{lim}\ f\ a)\ \mathsf{islim}_f\ a. \end{aligned} \quad (46)$$

Well-definedness follows from function comprehension and uniqueness. We shall formalize the type of lim later. Important for now are the following observations.

First, lim is a functional, not an ad hoc abstractor, and supports point-free expressions like $\mathsf{lim}\ f\ a$ (versus $\lim_{x \to a} f\ x$). Second, domain modulation covers left, right and two-sided limits, for example, given **def** $f : \mathbb{R} \to \mathbb{R}$ **with** $f\ x = (x \geq 0)\,?\,0 \nmid x + 1$, then $\mathsf{lim}\ f_{<0}\ 0 = 1$ and $\mathsf{lim}\ f_{\geq 0}\ 0 = 0$, whereas $0 \notin \mathcal{D}(\mathsf{lim}\ f_{\leq 0})$.

No separate concepts or notations are needed. Conventional notations can be synthesized from lim by macro definitions such that, for instance, if e is a real expression, $\lim_{x \to a} e$ stands for $\mathsf{lim}\,(x : \mathbb{R}\,.\,e)\,a$ and $\lim_{x \to <a} e$ stands for $\mathsf{lim}\,(x : \mathbb{R}_{<a}\,.\,e)\,a$ and so on.

4.2.2 *Calculating with Properties and Operators Regarding Orderings.* Orderings are perhaps the most important concept in programming and language semantics, and hence a good source of application examples for predicate calculus.

Definitions. For set X, define $\mathsf{pred}_X = X \to \mathbb{B}$ and $\mathsf{rel}_X = X^2 \to \mathbb{B}$. Table II defines well-known possible properties of relations in rel_X via a predicate $P : \mathsf{rel}_X \to \mathbb{B}$.

In the last line, $x\ ismin_{\prec}\ S \equiv x \in S \wedge \forall\, y : X\ .\ y \prec x \Rightarrow y \notin S$. We often write $\prec$ for R. Here $ismin\_$ had type $\mathsf{rel}_X \to X \times \mathcal{P}\, X \to \mathbb{B}$ but, as explained in Section 4.2.1 for adjacency, predicate transformers of type $\mathsf{rel}_X \to \mathsf{pred}_X \to \mathsf{pred}_X$ are more elegant. Hence we use the latter in the characterization of extremal elements in Table III.

Table II. Classification of Relations

Characteristic	P	Image, i.e., $P\,R \equiv$ formula below
reflexive	Refl	$\forall x : X \,.\, x\,R\,x$
irreflexive	Irfl	$\forall x : X \,.\, \neg(x\,R\,x)$
symmetric	Symm	$\forall (x, y) : X^2 \,.\, x\,R\,y \Rightarrow y\,R\,x$
asymmetric	Asym	$\forall (x, y) : X^2 \,.\, x\,R\,y \Rightarrow \neg(y\,R\,x)$
antisymmetric	Ants	$\forall (x, y) : X^2 \,.\, x\,R\,y \Rightarrow y\,R\,x \Rightarrow x = y$
transitive	Trns	$\forall (x, y, z) : X^3 \,.\, x\,R\,y \Rightarrow y\,R\,z \Rightarrow x\,R\,z$
total	Totl	$\forall (x, y) : X^2 \,.\, x\,R\,y \vee y\,R\,x$
equivalence	EQ	Trns $R \wedge$ Refl $R \wedge$ Symm R
preorder	PR	Trns $R \wedge$ Refl R
partial order	PO	PR $R \wedge$ Ants R
total order	TO	PO $R \wedge$ Totl R
quasi order	QO	Trns $R \wedge$ Irfl R (also called strict p.o.)
well-founded	WF	$\forall S : \mathcal{P}\,X \,.\, S \neq \emptyset \Rightarrow \exists x : X \,.\, x\,ismin_R\,S$

Table III. Characterization of Extremal Elements by Predicate Transformers

Name	*Symbol*	*Type*: $\mathsf{rel}_X \to \mathsf{pred}_X \to \mathsf{pred}_X$. *Image*: below
Lower bound	lb	$\mathsf{lb}_\prec\,P\,x \equiv \forall y : X \,.\, P\,y \Rightarrow x \prec y$
Least	lst	$\mathsf{lst}_\prec\,P\,x \equiv P\,x \wedge \mathsf{lb}\,P\,x$
Minimal	min	$\mathsf{min}_\prec\,P\,x \equiv P\,x \wedge \forall y : X \,.\, y \prec x \Rightarrow \neg(P\,y)$
Upper bound	ub	$\mathsf{ub}_\prec\,P\,x \equiv \forall y : X \,.\, P\,y \Rightarrow y \prec x$
Greatest	gst	$\mathsf{gst}_\prec\,P\,x \equiv P\,x \wedge \mathsf{ub}\,P\,x$
Maximal	max	$\mathsf{max}_\prec\,P\,x \equiv P\,x \wedge \forall y : X \,.\, x \prec y \Rightarrow \neg(P\,y)$
Least upper bound	lub	$\mathsf{lub}_\prec = \mathsf{lst}_\prec \circ \mathsf{ub}_\prec$
Greatest lower bound	glb	$\mathsf{glb}_\prec = \mathsf{gst}_\prec \circ \mathsf{lb}_\prec$

Educational experience shows that less formal characterizations cause beginning students to confuse *least*, *minimal* and *greatest lower bound* as being just different words for "at the lower side". This is aggravated by the fact that the well-foundedness axiom for natural numbers is stated in one textbook as "every nonempty subset has a minimal element", and in another as "every nonempty subset has a least element".

In fact, for any given relation $\prec$, no element can be both minimal and least, because $\neg(\mathsf{min}_\prec\,P\,x \wedge \mathsf{lst}_\prec\,P\,x)$; moreover $\mathsf{Refl}(\prec) \Rightarrow \neg(\mathsf{min}_\prec\,P\,x)$ and $\mathsf{Irfl}(\prec) \Rightarrow \neg(\mathsf{lst}_\prec\,P\,x)$. However, if we define a relation transformer $\overline{\prec} : \mathsf{rel}_X \to \mathsf{rel}_X$ with $x \mathrel{\overline{\prec}} y \equiv \neg(y \prec x)$, then $\mathsf{min}_{\overline{\prec}} = \mathsf{lst}_\prec$. If X is the set of naturals, $\mathsf{min}_< = \mathsf{lst}_\leq$, which clears up the issue.

4.2.2.1 *Calculational Reasoning about Extremal Elements.* In this example, we derive some properties used later. A predicate $P : \mathsf{pred}_X$ is called *monotonic* (or *upward closed* or *isotonic*) with respect to a relation $—\prec— : \mathsf{rel}_X$ according to the following definition.

$$\textit{Definition, Monotonicity.}\quad \forall (x, y) : X^2 \,.\, x \prec y \Rightarrow P\,x \Rightarrow P\,y. \tag{47}$$

This definition and most properties below are common knowledge in certain scientific communities and quite unknown in others. However, here the proofs are the important aspect, since they illustrate the similarity with the examples in

mathematical analysis, in particular how calculation makes proofs easy where semantic intuition offers no clue.

THEOREM, PROPERTIES OF EXTREMAL ELEMENTS. For any $—\prec—: \mathsf{rel}_X$ and $P : \mathsf{pred}_X$, (48)

(1) If $\prec$ is reflexive, then $\forall(y : X \,.\, x \prec y \Rightarrow P\,y) \Rightarrow P\,x$.
(2) If $\prec$ is transitive, then $\mathsf{ub}_\prec\, P$ is monotonic with respect to $\prec$.
(3) If P is monotonic with respect to $\prec$, then $\mathsf{lst}_\prec\, P\,x \equiv P\,x \;\wedge \forall(y : X \,.\, P\,y \;\equiv\; x \prec y)$.
(4) If $\prec$ is reflexive and transitive, then $\mathsf{lub}_\prec\, P\,x \equiv \forall(y : X \,.\, \mathsf{ub}\,P\,y \equiv x \prec y)$.
(5) If $\prec$ is antisymmetric, then $\mathsf{lst}_\prec\, P\,x \wedge \mathsf{lst}_\prec\, P\,y \Rightarrow x = y$ (uniqueness).

Replacing lb by ub and so on yields dual theorems (straightforward).

PROOF (OR OUTLINE). For part (1), instantiate the antecedent with $y := x$. For part (2), we assume $\prec$ transitive and prove $x \prec y \Rightarrow \mathsf{ub}_\prec\, P\,x \Rightarrow \mathsf{ub}_\prec\, P\,y$ in shunted form.

$$\begin{array}{lll}
\mathsf{ub}_\prec\, P\,x & \Rightarrow \langle p \Rightarrow q \Rightarrow p\rangle & x \prec y \Rightarrow \mathsf{ub}_\prec\, P\,x \\
& \equiv \langle \text{Definition ub}\rangle & x \prec y \Rightarrow \forall z : X \,.\, P\,z \Rightarrow z \prec x \wedge 1 \\
& \equiv \left\langle p \Rightarrow e\begin{bmatrix} v \\ 1 \end{bmatrix} = e\begin{bmatrix} v \\ p \end{bmatrix}\right\rangle & x \prec y \Rightarrow \forall z : X \,.\, P\,z \Rightarrow z \prec x \wedge x \prec y \\
& \Rightarrow \langle \text{Transitiv. } \prec\rangle & x \prec y \Rightarrow \forall z : X \,.\, P\,z \Rightarrow z \prec y \\
& \equiv \langle \text{Definition ub}\rangle & x \prec y \Rightarrow \mathsf{ub}_\prec\, P\,y\,.
\end{array}$$

For part (3), we assume P monotonic and calculate $\mathsf{lst}_\prec\, P\,x$

$$\begin{array}{ll}
\mathsf{lst}_\prec\, P\,x & \\
\equiv \langle \text{Defin. lst, lb}\rangle & P\,x \wedge \forall\, y : X \,.\, P\,y \Rightarrow x \prec y \\
\equiv \langle \text{Modus Pon.}\rangle & P\,x \wedge (P\,x \Rightarrow \forall\, y : X \,.\, P\,y \Rightarrow x \prec y) \\
\equiv \langle \text{L-dstr. } \Rightarrow/\forall\rangle & P\,x \wedge \forall\, y : X \,.\, P\,x \Rightarrow P\,y \Rightarrow x \prec y \\
\equiv \langle \text{Monoton. } P\rangle & P\,x \wedge \forall\, y : X \,.\, (P\,x \Rightarrow P\,y \Rightarrow x \prec y) \wedge (P\,x \Rightarrow x \prec y \Rightarrow P\,y) \\
\equiv \langle \text{L-dstr. } \Rightarrow/\wedge\rangle & P\,x \wedge \forall\, y : X \,.\, P\,x \Rightarrow (P\,y \Rightarrow x \prec y) \wedge (x \prec y \Rightarrow P\,y) \\
\equiv \langle \text{Mut. implic.}\rangle & P\,x \wedge \forall\, y : X \,.\, P\,x \Rightarrow (P\,y \equiv x \prec y) \\
\equiv \langle \text{L-dstr. } \Rightarrow/\forall\rangle & P\,x \wedge (P\,x \Rightarrow \forall\, y : X \,.\, P\,y \equiv x \prec y) \\
\equiv \langle \text{Modus Pon.}\rangle & P\,x \wedge \forall(y : X \,.\, P\,y \equiv x \prec y)
\end{array}$$

Some variations in this proof are outlined in [Boute 2004]. Furthermore, part (4) is a direct consequence from the preceding parts. Proving part (5) is simple but very typical.

$$\begin{array}{ll}
\mathsf{lst}_\prec\, P\,x \wedge \mathsf{lst}_\prec\, P\,y & \\
\equiv \langle \text{Defin. lst, lb}\rangle & P\,x \wedge \forall(y : X \,.\, P\,y \Rightarrow x \prec y) \wedge P\,y \wedge \forall(x : X \,.\, P\,x \Rightarrow y \prec x) \\
\Rightarrow \langle \text{Instantiation}\rangle & P\,x \wedge (P\,y \Rightarrow x \prec y) \wedge P\,y \wedge (P\,x \Rightarrow y \prec x) \\
\equiv \langle \text{Rearranging}\rangle & P\,x \wedge (P\,x \Rightarrow y \prec x) \wedge P\,y \wedge (P\,y \Rightarrow x \prec y) \\
\equiv \langle \text{Modus Pon.}\rangle & P\,x \wedge y \prec x \wedge P\,y \wedge x \prec y \\
\Rightarrow \langle \text{Weakening}\rangle & y \prec x \wedge x \prec y \\
\Rightarrow \langle \text{Antisymm.}\rangle & x = y
\end{array}$$

These properties are crucial in programming and in formal language semantics.

As shown for limits, we use the predicate transformers to define functionals $\sqcap$ (and $\sqcup$) mapping X-valued functions to the glb (and lub) of their range in case existence and uniqueness are satisfied. The relevant predicate is the *range predicate* $\mathsf{R}\_ : (Y \to X) \to X \to \mathbb{B}$ defined by $\mathsf{R}_f\, x \equiv x \in \mathcal{R}\, f$. This allows defining infix operators like $\sqcup$ by $x \sqcup y = \sqcup(x, y)$ by variadic application (defined in Section 2.2.2).

Furthermore, when appropriate, X is extended by adding $\top$ (top) and $\bot$ (bottom) elements, as in any textbook on lattice theory [Gierz et al. 1980] or formal semantics [Loeckx and Sieber 1984; Stoy 1977; Winskel 1993].

4.2.2.2 *Brief Note on Induction.* Relation $\prec : X^2 \to \mathbb{B}$ *supports induction* iff $\mathsf{SI}(\prec)$, where

$$\mathsf{SI}(\prec) \;\equiv\; \forall P : \mathsf{pred}_X \,.\, \forall (x : X \,.\, \forall (y : X_{\prec x} \,.\, P\, y) \Rightarrow P\, x) \Rightarrow \forall P \qquad (49)$$

One can show $\mathsf{SI}(\prec) \equiv \mathsf{WF}(\prec)$; a calculational proof is given in [Boute 2002]. Examples are the familiar strong and weak induction over $\mathbb{N}$, where one of the axioms is that every nonempty subset has a *least* element under $\leq$ (or *minimal* element under $<$). They can be obtained by taking for $\prec$ respectively $<$ and $\prec$ with $m \prec n \equiv m + 1 = n$.

$$\text{Strong Induction.}\ \forall P : \mathsf{pred}_{\mathbb{N}} \,.\, \forall (n : \mathbb{N} \,.\, \forall (m : \mathbb{N} \,.\, m < n \Rightarrow P\, m) \Rightarrow P\, n) \Rightarrow \forall P,$$
$$\text{Weak Induction.}\ \forall P : \mathsf{pred}_{\mathbb{N}} \,.\, P\, 0 \wedge \forall (n : \mathbb{N} \,.\, P\, n \Rightarrow P\, (n+1)) \Rightarrow \forall P.$$

A later example is structural induction over data structures.

An important preparatory step to avoid errors in inductive proofs is always making the predicate P and all quantification explicit, avoiding vague designations such as "induction over n". This is especially important in case other variables are involved.

Remark. This section showed the advantages of formal predicate calculus over the usual informal style in exposition. The case study next illustrates exploration.

4.2.3 *A Case Study: When Is a Greatest Lower Bound a Least Element?* The next few theorems, if reported at all in the literature, were unknown to the author before deriving them calculationally. Hence, they illustrate well how the formal rules serve as an instrument for discovery when entering an area where one feels uncertain at first. Indeed, general orderings being rather abstract, examples are not only difficult to construct, but may also have hidden properties not covered by the assumptions.

The motivation arose in a minitheory for recursion (given later) as follows. The g.l.b. operator $\wedge$, being the particularization of $\sqcap$ to $\mathbb{R}' := \mathbb{R} \cup \{-\infty, +\infty\}$ under $\leq$, is important in analysis. The recursion theory required least elements of nonempty subsets of $\mathbb{N}$. For these, it seems "intuitive", for example, from a picture as in Figure 1, that *both concepts coincide*. This would imply that a separate "least" operator is avoidable (contrary to first impression, we do not like introducing operators without necessity).

However, is this really obvious? The diagram provides no clue as to which axioms of $\mathbb{N}, \leq$ are involved, and hence is useless for generalization. So the

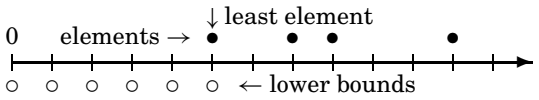


Fig. 1. Least element and greatest lower bound for natural numbers.

exercise consisted in formally proving this property, not for verifying it, but for uncovering the more basic properties of natural numbers that give rise to it, and finding generalizations.

The linear diagram suggests totality as the basic property. This is misleading, since $\mathbb{R}$ is totally ordered, yet $P : \mathbb{R} \to \mathbb{B}$ with $\forall x : \mathcal{D}\ P\ .\ P\ x \equiv x \neq 0 \wedge 1/x \in \mathbb{N}$ has g.l.b. 0 but no least element. Proof details are given in [Boute 2004], after showing $\mathsf{lb}_{\leq}\ P\ x \equiv x \leq 0$.

This negative result (a g.l.b. but no least element) is a standard example in analysis courses, but here we want to know which properties of $\mathbb{N}$ yield positive results. A major difference with $\mathbb{R}$ is well foundedness, which therefore is expected to play a role.

Yet, in proving certain theorems for $\mathbb{N}$ with $\leq$, it was found that no assumptions were needed, and hence such theorems will be restated here in general form. Moreover, another useful by-product of certain proofs was the following general-purpose rule.

THEOREM, QUANTIFIED SHUNTING. For $P : X \to \mathbb{B}$, $Q : Y \to \mathbb{B}$, $—R— : X \times Y \to \mathbb{B}$, $\forall (x : X\ .\ P\ x \Rightarrow \forall (y : Y\ .\ Q\ y \Rightarrow x\ R\ y)) \equiv \forall (y : Y\ .\ Q\ y \Rightarrow \forall (x : X\ .\ P\ x \Rightarrow x\ R\ y))$

PROOF. By left distributivity of $\Rightarrow/\forall$, shunting and dummy swap. □

This pattern arose in (and is now factored out of) the proof of the following lemma.

LEMMA. $\forall (x : X\ .\ P\ x \Rightarrow \forall\ y : X\ .\ \mathsf{lb}_{\prec}\ P\ y \Rightarrow\ y \prec x)$ (*parentheses for clarity only*)

PROOF. Weaken the definition $\forall (x : X\ .\ \mathsf{lb}_{\prec}\ P\ x \equiv \forall\ y : X\ .\ P\ y \Rightarrow x \prec y)$ from $\equiv$ to $\Rightarrow$, apply quantified shunting and rename dummies. □

An immediate application is the following.

THEOREM, LEAST ELEMENTS AS GLBS. $\mathsf{lst}_{\prec}\ P\ x \equiv P\ x \wedge \mathsf{glb}_{\prec}\ P\ x$.

PROOF

$$\begin{array}{rcll} \mathsf{lst}_{\prec}\ P\ x & \equiv & \langle \text{Def. lst, lemma} \rangle & P\ x \wedge \mathsf{lb}_{\prec}\ P\ x \wedge (P\ x \Rightarrow \forall\ y : X\ .\ \mathsf{lb}_{\prec}\ P\ y \Rightarrow\ y \prec x) \\ & \equiv & \langle \text{Modus Ponens} \rangle & P\ x \wedge \mathsf{lb}_{\prec}\ P\ x \wedge \forall\ y : X\ .\ \mathsf{lb}_{\prec}\ P\ y \Rightarrow\ y \prec x \\ & \equiv & \langle \text{Definition glb} \rangle & P\ x \wedge \mathsf{glb}_{\prec}\ P\ x \end{array}$$

As a corollary, $\mathsf{lst}_{\prec}\ P\ x \Rightarrow \mathsf{glb}_{\prec}\ P\ x$. We saw that the converse does not generally hold. In checking that it holds for $\mathbb{N}$ with $\leq$, we introduce known properties of $\mathbb{N}$, but as weak as possible, yet enabling the next step. Guidance is given by the formal rules.

THEOREM, GLBS AS LEAST ELEMENTS (FOR $\mathbb{N}$). $\mathsf{glb}_{\leq}\ P\ n\ \Rightarrow P\ n\ \Rightarrow \mathsf{lst}_{\leq}\ P\ n$. (50)

PROOF. Since the relation and the predicate are fixed, we use shorthands L for $\mathsf{lb}_{\leq} P$ and K for $\mathsf{lst}_{\leq} P$ and I (infimum) for $\mathsf{glb}_{\leq} P$ to reduce handwriting.

$$\begin{array}{lll}
I\,n \Rightarrow & \langle\text{Weakening}\rangle & \forall\, m:\mathbb{N}\ .\ L\,m \Rightarrow m \leq n \\
\equiv & \langle\text{Contraposit}\rangle & \forall\, m:\mathbb{N}\ .\ n < m \Rightarrow \neg L\,m \\
\equiv & \langle\text{Definition } L\rangle & \forall\, m:\mathbb{N}\ .\ n < m \Rightarrow \neg\forall\, k:\mathbb{N}\ .\ P\,k \Rightarrow m \leq k \\
\equiv & \langle\text{Duality } \forall/\exists\rangle & \forall\, m:\mathbb{N}\ .\ n < m \Rightarrow \exists\, k:\mathbb{N}\ .\ P\,k \wedge k < m \\
\Rightarrow & \langle\text{Weakening}\rangle & \forall\, m:\mathbb{N}\ .\ n < m \Rightarrow \exists\, P \\
\equiv & \langle\text{R-dstr. } \Rightarrow/\exists\rangle & \exists\,(m:\mathbb{N}\ .\ n < m) \Rightarrow \exists\, P \\
\equiv & \langle\text{Property } \alpha\rangle & \exists\, P \\
\Rightarrow & \langle\text{Property } \beta\rangle & \exists\, K \\
I\,n \equiv & \langle I\,n \Rightarrow \exists\, K\rangle & \exists\, K \wedge I\,n \\
\Rightarrow & \langle\text{Property } \gamma\rangle & \exists\, K \wedge \forall\, m:\mathbb{N}\ .\ I\,m \Rightarrow m = n \\
\Rightarrow & \langle K\,n \Rightarrow I\,n\rangle & \exists\, K \wedge \forall\, m:\mathbb{N}\ .\ K\,m \Rightarrow m = n \\
\Rightarrow & \langle\text{Half-pt. rule}\rangle & K\,n
\end{array}$$

Property α is $\forall\, n:\mathbb{N}.\ \exists\, m:\mathbb{N}.\ n < m$, which is fairly weak. Property β is just the predicative formulation of well foundedness, namely $\forall\, P:\mathbb{N}\to\mathbb{B}.\ \exists\, P \Rightarrow \exists(\mathsf{lst}_{\leq} P)$, which may also be written $\exists\, P \equiv \exists(\mathsf{lst}_{\leq} P)$ since $\mathsf{lst}_{\leq} P\,n \Rightarrow P\,n$. Property γ is antisymmetry of $\leq$, with consequence $\forall\, n:\mathbb{N}\ .\ I\,n \Rightarrow \forall\, m:\mathbb{N}\ .\ I\,m \Rightarrow m = n$. □

For the general case, we separate the concerns, in this case, the assumptions.

Observe that, in case $\neg\exists\, P$ (or $P\,x \equiv 0$), one has $\mathsf{lst}_{\prec} P\,x \equiv 0$ but $\mathsf{lb}_{\prec} P\,x \equiv 1$, hence $\mathsf{glb}_{\prec} P\,x \equiv \forall\, y:X\ .\ y \prec x$. For having $\mathsf{glb}_{\prec} P = \mathsf{lst}_{\prec} P$, a necessary and sufficient condition is $\forall\, x:X\ .\ \mathsf{glb}_{\prec} P\,x \equiv 0$, or $\forall\, x:X\ .\ \exists\, y:X\ .\ \neg(y \prec x)$, generalizing property α. Hence, unless we accept $\neg\exists\, P$ as an exception, this condition is necessary.

Since many orderings of interest in computing are antisymmetric, it is worthwhile investigating how far this leads. One result is the lemma

LEMMA. *If $\prec$ is antisymmetric,* $\exists(\mathsf{lst}_{\prec} P) \Rightarrow \mathsf{glb}_{\prec} P\,x \Rightarrow \mathsf{lst}_{\prec} P\,x$.

PROOF (SHUNTED)

$$\begin{array}{lll}
\mathsf{glb}_{\prec} P\,x \Rightarrow & \langle\text{Antisymmetry}\rangle & \forall\, y:X\ .\ \mathsf{glb}_{\prec} P\,y \Rightarrow x = y \\
\Rightarrow & \langle \mathsf{lst}_{\prec} P \mathrel{\hat{\ni}} \mathsf{glb}_{\prec} P\rangle & \forall\, y:X\ .\ \mathsf{lst}_{\prec} P\,y \Rightarrow x = y \\
\Rightarrow & \langle\text{Half-pint rule}\rangle & \exists(\mathsf{lst}_{\prec} P) \Rightarrow \mathsf{lst}_{\prec} P\,x \quad \square
\end{array}$$

Combining this with the property $\mathsf{lst}_{\prec} P\,x \Rightarrow \mathsf{glb}_{\prec} P\,x$ obtained earlier yields.

THEOREM. *If $\prec$ is antisymmetric, then* $\exists(\mathsf{lst}_{\leq} P) \Rightarrow \mathsf{glb}_{\prec} P = \mathsf{lst}_{\prec} P$*;*
if $\prec$ is antisymmetric and well founded, then $\exists\, P \Rightarrow \mathsf{glb}_{\prec} P = \mathsf{lst}_{\prec} P$.

Observe that the argument has become shorter than the first proof for $I\,n \Rightarrow K\,n$, which still remains interesting to reflect the flavor of a first attempt.

Without further elaboration, we conclude by generalizing $\wedge$ from (8) to the g.l.b operator $\bigwedge$ mapping any $\mathbb{R}$-valued function f to an element of $\mathbb{R}'$, with

$$\forall\, x:\mathbb{R}'\ .\ \bigwedge f = x \equiv \forall(y:\mathcal{R}\ f\ .\ x \leq y) \wedge \forall\, z:\mathbb{R}'\ .\ (\forall\, y:\mathcal{R}\ f\ .\ z \leq y) \Rightarrow z \leq x \quad (51)$$

and likewise for $\bigvee$. Let $P:\mathbb{N}\to\mathbb{B}$ and $g := \bigwedge n:\mathbb{N} \mid P\,n$, then $\neg\exists\, P \equiv g = \infty$ and $\exists\, P \equiv g \in \mathbb{N} \wedge P\,g \wedge \forall\, m:\mathbb{N}\ .\ m < g \Rightarrow \neg P\,m$, linking $\bigwedge$ to the least element.

5. INTERMEZZO: DECLARATIVE LANGUAGE PRAGMATICS

The simple design described in Section 2.2 has a wide applicability, sufficient to cover most, if not all, mathematics needed in engineering and computing. Precisely because of its smallness, the syntax alone gives few clues about how to do this. This situation is reminiscent of LISP as compared to more baroque programming languages. Claiming wide applicability while leaving readers to try it all out in their own field is not sufficient. Therefore, in this and the following sections, we address successively the language design aspects that validate the claim, provide a general-purpose operator package or "user library", and a fair number of examples in diverse application domains.

5.1 Pragmatics of Language Constructs and Operator Design

5.1.1 *Some Additional Notes on the Language Constructs*

5.1.1.1 *Identifiers.* As explained in Section 2.2 and illustrated in many examples, the bound variables (dummies) in an *abstraction* (*binding* . *expression*) have *local* scope, and their names can be changed (so-called α-conversion) in the manner familiar to anyone who has handled multiple sums or integrals and formalized in lambda calculus [Barendregt 1984; Stoy 1977].

Constants introduced in a *definition* (**def** *binding*) have *global* scope. A definition **def** $a : X$ **with** p states the axiom $a \in X \wedge p$ for a and asserts *existence*, viz., $\exists x : X \,.\, p[^a_x$ and *uniqueness*, viz., $\forall (x, y) : X^2 \,.\, p[^a_x \wedge p[^a_y \Rightarrow x = y$, which are proof obligations. For explicit definitions, as in most examples in this paper, these are met trivially.

We briefly comment on functions. In definitions **def** $f : X$ **with** $f\ v = e$, the **with** part stands for $\forall v : \mathcal{D}\ f \,.\, f\ v = e$ (this convention can be generalized and made sufficiently precise, even for automation). For explicit image definitions (**with** $f\ v = e$ where $f \notin \varphi\ e$), the proof obligations are met trivially. For implicit definitions, as shown for $\lim$ in (46) and $\curlywedge$ in (51), they are met by function comprehension (37) and a uniqueness proofs; the conditions are included in the domain specification. A systematic discipline for organizing this is the *trilogy principle* in [Boute 2002].

A variant is *specification*, which has the form **spec** *binding* and states an axiom without existence or uniqueness claims. A last variant is the *local definition* of the form *expression* **where** *binding*, which is the same as **def** but with local scope. Further mechanisms for packaging and import/export can be freely borrowed from elsewhere.

In contexts containing many definitions with common conventions (as in many places in this article), a global legend (with metavariables) from Section 2.2.1 obviates repetitive type definitions that **def**-definitions would entail.

5.1.1.2 *Tupling.* For various reasons, tupling deserves a separate construct rather than some simulation by lambda terms. First, lambda terms impose Currying. In most of mathematics, the Cartesian product is a central concept and Currying is *not* always wanted. Second, mimicking our functional tuples in lambda calculus requires at least the following. Assuming Church numerals

$\gamma\, n = \lambda x\, y\, .x^n\, y$, one must design two combinators: a *tuple constructor* **M** specified by $\mathbf{M}(\gamma\, k)\, N_0 \cdots N_{k-1}\, (\gamma\, i) = N_i$, where the N_i are any terms $(0 \leq i < k)$, and a *length* operator **L** such that $\mathbf{L}\,(\mathbf{M}(\gamma\, k)\, N_0 \cdots N_{k-1}) = \gamma\, k$ (an instructive exercise). Third, even such simulation remains too clumsy for practical use (try translating $f\,(x,\, y) = x^2 + y^2$) and synthesizing common conventions. Finally, handling tuples as first-class-functions is better served by a dedicated construct.

Our tuples also differ from those in functional programming. In most programming languages, even functional ones, tuples and lists are mutually distinct concepts and are *not functions*. For instance, in Haskell [Hudak et al. 1999] tuples are just aggregates of the form `(x,y,z)`, and lists are defined recursively from the empty list `[]` and `:`, writing `[x,y,z]` for `x:y:z:[]`. The same holds in languages for proof assistants like Isabelle [Paulson 2001].

By contrast, in Funmath, we unify tuples, sequences, lists, etc. as first-class *functions* with all privileges, such as appearing as arguments of generic functionals and variadic operators. Due to the evident isomorphism between Funmath tuples and the variety of tupling and list constructs in programming languages, all calculations in our formalism are relevant to these constructs *even in languages not supporting them as functions*.

5.1.2 *Elastic Operators, Variadic Infix Application and Design Issues*

5.1.2.1 *Elastic Operators.* Common mathematics contains many ad hoc abstractors such as $\lim_{x \to a}$, $\{x : X \mid \ldots\}$, $\sum_{i=0}^{k-1}$ and $(\forall\, x)$. We replace these in a uniform way by so-called *elastic operators*, that is, functionals such as lim, $\mathcal{R}$, $\sum$, $\forall$, applied to functions.

Advantages are: (a) supporting point-free expression, as in $\mathsf{lim}\, f\, a, \mathcal{R}\, f, \sum f$, $\forall P$; (b) pointwise expression requires only one kind of abstraction (function abstraction), providing separation of concerns in the calculation rules; (c) the *domain modulation* principle mentioned in Section 4.1.1 together with function filtering (19) yields finer expressivity than the "range" associated with ad hoc abstractors.

Given the variety of existing conventions, some abstractor notations are bound to be textually identical, as in Nuprl [1992] but the different parsing must be observed: in Funmath, $\mathrm{E}\, v : X\, .\, e$ is the application of a function E (elastic operator) to a function $v : X\, .\, e$, as opposed to an ad hoc abstractor "$\mathrm{E}\, v : X$." before an expression e. Moreover, our approach differs from other unifications of ad hoc abstractors [Dijkstra and Scholten 1990; Gries and Schneider 1993] by its functional nature, by supporting both point-free and pointwise styles, by not being restricted to associative and commutative operators, and by applicability to tuples.

For instance, the so-called Eindhoven Quantifier Notation $\langle Q\, x : p.x : f.x \rangle$ is uniform in the sense that Q can be the symbol $\forall$, Σ, S (for sets), but this is not an operator, and each abstraction is monolithic. This precludes sharing properties with functions such as the calculation rules and the generic functionals defined later. The compactness of point-free expressions can be achieved only by ad hoc conventions, for example, $[t]$ for $\langle \forall z :: t.z \rangle$ and $\langle t \rangle$ for $\langle \exists z :: t.z \rangle$, as in

Table IV. Examples of Variadic Infix Application

Expression	*Traditional justification*	*Expression*		*Funmath justification*
$x + y + z$	associativity,	$x + y + z$	$=$	$\sum(x, y, z)$
$x \vee y \vee z$	associativity	$x \vee y \vee z$	$\equiv$	$\exists(x, y, z)$
$x \,\{\}\, y \,\{\}\, z$	(not considered)	$x \,\{\}\, y \,\{\}\, z$	$=$	$\{x, y, z\}$
$X \times Y \times Z$	ad hoc convention	$X \times Y \times Z$	$=$	$\bigtimes(X, Y, Z)$
$x = y = z$	conjunctionality	$x = y = z$	$\equiv$	$\mathsf{con}(x, y, z)$
$x \neq y \neq z$	(not considered)	$x \neq y \neq z$	$\equiv$	$\mathsf{inj}(x, y, z)$

[Dijkstra 1996b]. By contrast, any of our elastic operators E simply yields $\mathrm{E}\, f = \mathrm{E}\, x : \mathcal{D}\ f\ .\ f\ x$ (Leibniz's principle), as in $\forall\, P \equiv \forall\, x : \mathcal{D}\ P\ .\ P\ x$.

For $\forall$ and $\exists$, a similar formulation as predicates about the constancy of predicates is found in HOL Light [Harrison 2000], but since HOL Light abstractions are untyped, essential concepts such as domain modulation are not supported (and neither is variadic $\wedge$, $\vee$).

5.1.2.2 *Variadic Infix Application.* Tuple arguments for elastic operators like $\forall$, $\mathcal{R}$ and $\sum$ may have any length. Generally, operators whose arguments are tuples of any length are called *variadic* [Illingworth et al. 1989], for example, restrictions of elastic operators to tuples in their domain.

Argument/operator alternations like $x \star y \star z$ (any length) are ubiquitous in mathematics, but provided with various ad hoc justifications, as shown in the leftmost half of Table IV. *Associativity* for an operator $\_ \star \_ : A^2 \to A$ means $x \star (y \star z) = (x \star y) \star z$ and *conjunctionality* for a relation $\_R\_ : A^2 \to \mathbb{B}$ means $x\ R\ y\ R\ z \equiv x\ R\ y \wedge y\ R\ z$.

Instead of resorting to such disparate ad hoc justifications, we uniformly define *variadic infix application* via the application of an appropriate variadic operator F, viz., $x \star y \star z = F\ (x, y, z)$, as shown in the rightmost half of Table IV. Like the elastic operator con from (38), the elastic inj (*injective*) is a predicate on functions, with

$$\mathsf{inj}\ f \quad \equiv \quad \forall\, x : \mathcal{D}\ f\ .\ \forall\, y : \mathcal{D}\ f\ .\ f\ x = f\ y \Rightarrow x = y\,. \tag{52}$$

The last four rows illustrate uniform treatment of operators that are not commutative or not associative. The "superconjunctionality" in the last row gives $x \neq y \neq z$ its most useful purpose (x, y, z distinct), which traditional conventions cannot achieve.

5.1.2.3 *Design Issues.* Formally, $F : D \to Y$ is an *elastic extension* of $\_ \star \_ : X^2 \to Y$ iff $X^* \subseteq D \subseteq \mathsf{fam}\, X \wedge \forall\, x, x' : X^2\ .\ F\ (x, x') = x \star x'$. The *family* operator fam is defined by

$$f \in \mathsf{fam}\ X \equiv \mathcal{R}\ f \subseteq X\,, \tag{53}$$

and $f \in \mathsf{fam}\, \mathbb{Z}$ is read: "f is a family of integers". The most elastic case is $D = \mathsf{fam}\, X$, but this is not achievable for all $\star$. It is for quantifiers, noting that $\forall\, (x, x') \equiv x \wedge x'$. Elastic extensions are not unique, leaving room for judicious design.

For instance, if $\star$ is associative, it is wise to honor common convention by choosing the elastic extension F such that $x \star y \star z = (x \star y) \star z$.

To that effect, $F \rceil X^*$ must be a *list homomorphism* [Bird 1998], viz., $\forall f, g : (X^*)^2 \,.\, F\,(f +\!\!+ g) = F\ f \star F\ g$, where $+\!\!+$ is concatenation. To appreciate this, it is worth designing such an elastic extension of $\equiv$. For elastic operators not related to pre-existing operators, design is unrestricted.

Our approach was first meant just to generalize Zamfirescu's [1993] design method for *resolution functions* in VHDL [IEEE 1994], and the name "elastic operator" for our solution was coined by Jacques Zahnd (personal communication, 1993). Wider applicability was soon apparent throughout "everyday" mathematics and in the design of formal mathematical languages. For instance, the approach solves all reported problems with variadic operators in OpenMath [Davenport 2000].

5.2 Generic Functionals

Predicate calculus allows expanding the library of generic functionals started in Section 2.3.3, with the same design principle of avoiding restrictions on the argument functions. A small number suffices, even for practical purposes and not just in the theoretical sense whereby $\boldsymbol{K}$ and $\boldsymbol{S}$ suffice in combinator calculus [Barendregt 1984]. They can be seen as typed variants of combinators, but typing is crucial to making them practical, since it enables specifying the result domains according to the design principle. They possess interesting algebraic properties, forming in a concrete variant of category theory [Aarts et al. 1992; Reynolds 1980].

We adopt the discipline of keeping generalizations of existing operators *conservative* in that they specialize to the common definitions in case the usual restrictions are satisfied. In this way, no familiar properties are lost. Otherwise, we reserve all freedom.

5.2.1 *Auxiliary Operators for Function Typing.* The following generic operator is designed later but defined here so the reader can use it to specify the types of other generic functionals in Section 5.2.2 which is a most useful familiarization exercise. We define $\bigtimes$ as follows: for any set-valued function T,

$$\bigtimes T \;=\; \{f : \mathcal{D}\,T \to \bigcup T \mid \forall x : \mathcal{D}\,f \cap \mathcal{D}\,T \,.\, f\,x \in T\,x\}. \tag{54}$$

By calculation, $\bigtimes(X, Y) = X \times Y$ and $\bigtimes(X \bullet Y) = X \to Y$. We introduce $X \ni x \to Y$ (while allowing $x \in \varphi Y$) as a macro for $\bigtimes x : X \,.\, Y$, which is convenient for chained dependencies, as in $X \ni x \to Y \ni y \to Z$ (allowing $x \in \varphi(Y, Z)$ and $y \in \varphi Z$).

We define the *block* operator over $\mathbb{N}' := \mathbb{N} \cup_\iota \infty$ as follows: for any $n : \mathbb{N}'$,

$$\square n \;=\; \{m : \mathbb{N} \mid m < n\}. \tag{55}$$

Examples. $\square 0 = \emptyset$ and $\square 2 = \mathbb{B}$ and $\square \infty = \mathbb{N}$. We define the *power* of a set X by $X \uparrow n \;=\; \square n \to X$ (shorthand: X^n) and (re)define Kleene's $*$ by $X^* = \bigcup n : \mathbb{N} \,.\, X^n$.

5.2.2 *A Package of Generic Functionals*

5.2.2.1 *Basic Functionals.* For explicit definitions, we use abstraction, which is compact and keeps domain and mapping together during calculations:

handling both simultaneously yields shorter and more synoptic derivations. We present the operators in groups.

(i) *Operators from Functions to Sets*. *domain* $\mathcal{D}$ (specified in the binding), *range* $\mathcal{R}$ (40), *graph* $\mathcal{G}$ (56), *bijective domain* Bdom (57) and *bijective range* Bran (58).

$$\mathcal{G}\ f \ = \ \{x,\ f\ x \mid x : \mathcal{D}\ f\} \tag{56}$$

$$\mathsf{Bdom}\ f \ = \ \{x : \mathcal{D}\ f \mid \forall\, y : \mathcal{D}\ f\ .\ f\ x = f\ y \Rightarrow x = y\} \tag{57}$$

$$\mathsf{Bran}\ f \ = \ \{f\ x \mid x :\ \mathsf{Bdom}\ f\} \tag{58}$$

(ii) *Operators for Function Typing*. *function arrow* $\rightarrow$ (24), *partial arrow* $\nrightarrow$ (25), *family type* fam (53) and *funcart product* $\bigtimes$ (54).

(iii) *Function Transformer with One Function Argument*. *function inverse* $—^{-}$.

$$f^{-} \in \mathsf{Bran}\ f \rightarrow\ \mathsf{Bdom}\ f \quad \text{and} \quad \forall x :\ \mathsf{Bdom}\ f\ .\ f^{-}(f\ x) = x. \tag{59}$$

To remove doubts: (59) does not assume inj f, but yields the usual inverse in case inj f.

(iv) *Function Transformer with One Function and One Predicate Argument*. *filter* $\downarrow$ (19); function transformer with one function and one set argument: *restrict* $\restriction$ (20).

(v) *Function Transformers with Two Function Arguments*. *composition* $\circ$ (13), *dispatching* using Meyer's [1991] symbol & (60), *parallel* $\parallel$ (61), *override* $\oslash$ (62), *overridden* $\otimes$ (63), and *function merge* $\cup$ (18). Obviously, $f \mathbin{©} g \Rightarrow f \cup g = f \otimes g = f \oslash g$.

$$f\ \&\ g \ = \ x : \mathcal{D}\ f \cap \mathcal{D}\ g\ .\ f\ x, g\ x \tag{60}$$

$$f \parallel g \ = \ (x, y) : \mathcal{D}\ f \times \mathcal{D}\ g\ .\ f\ x, g\ y \tag{61}$$

$$f \oslash g \ = \ x : \mathcal{D}\ f \cup \mathcal{D}\ g\ .\ (x \in \mathcal{D}\ f)\,?\ f\ x \nmid g\ x \tag{62}$$

$$f \otimes g \ = \ x : \mathcal{D}\ f \cup \mathcal{D}\ g\ .\ (x \in \mathcal{D}\ g)\,?\ g\ x \nmid f\ x \tag{63}$$

(vi) *Function Transformers for* direct extension *in Several Variants*. *monadic* $=\!\!=$ (14), *dyadic* $\widehat{=}$ (15), *left half* $\leftharpoonup$ and *right half direct extension* $\rightharpoonup$ (16).

(vii) *Predicates with One Function Argument*. *constant* con (38), *injective* inj (52), *surjective* srj (64), *bijective* bij (65), *finite* fin (66) and with two function arguments (i.e., relational): *compatibility* © (21), *function equality* (23), *subfunction* $\subseteq$ (67)

$$f\ \mathsf{srj}\ Y \ \equiv \ \mathcal{R}\ f = Y \tag{64}$$

$$f\ \mathsf{bij}\ Y \ \equiv \ \mathsf{inj}\ f \wedge f\ \mathsf{srj}\ Y \tag{65}$$

$$\mathsf{fin}\ f \ \equiv \ \exists n : \mathbb{N}\ .\ \exists g : \square n \rightarrow \mathcal{D}\ f\ .\ g\ \mathsf{srj}\ \mathcal{D}\ f. \tag{66}$$

$$f \subseteq g \ \equiv \ f = g \restriction \mathcal{D}\ f \tag{67}$$

Clearly, $\subseteq$ is a partial ordering on functions, and coincides with the set ordering on the function graphs, hence reading $\subseteq$ from the usual "functions as sets" view is harmless.

5.2.2.2 *Elastic Extensions of Two-Argument Generic Functionals.* We only show extensions that are "interesting" in that their arguments can be fully arbitrary families of functions, *not necessarily with discrete domains*. The main operators are: *transpose* $—^{\mathsf{T}}$ (68), which extends & since $(f, g)^{\mathsf{T}} = f \,\&\, g$, and its "uniting variant" $—^{\mathsf{U}}$ (69), *elastic parallel* $\|$ (70), *elastic direct extension* $\preceq$(71), *elastic compatibility* Ⓒ (72): for any family of functions F and any function g (in practice an elastic operator),

$$F^{\mathsf{T}} = y : \bigcap(\mathcal{D} \circ F) \,.\, x : \mathcal{D}\, F \,.\, F\, x\, y \tag{68}$$

$$F^{\mathsf{U}} = y : \bigcup(\mathcal{D} \circ F) \,.\, x : \mathcal{D}\, F \mathbin{\wedge} y \in \mathcal{D}(F\, x) \,.\, F\, x\, y \tag{69}$$

$$\|F f = x : \mathcal{D}\, F \cap \mathcal{D}\, f \mathbin{\wedge} f\, x \in \mathcal{D}(F\, x) \,.\, F\, x\, (f\, x) \tag{70}$$

$$\overset{\preceq}{g} F = g \circ F^{\mathsf{T}} \tag{71}$$

$$Ⓒ\, F \equiv \forall (x, y) : (\mathcal{D}\, F)^2 \,.\, F\, x \,Ⓒ\, F\, y. \tag{72}$$

Noteworthy is also the *elastic function merge* $\bigcup$, defined by domain axiom $\mathcal{D}(\bigcup F) = \{y : \bigcup(\mathcal{D} \circ F) \mid \forall (x, x') : (\mathcal{D}\, F)^2 \,.\, y \in \mathcal{D}(F\, x) \cap \mathcal{D}(F\, x') \Rightarrow F\, x\, y = F\, x'\, y\}$ and mapping $y \in \mathcal{D}(\bigcup F) \Rightarrow \forall x : \mathcal{D}\, F \,.\, y \in \mathcal{D}(F\, x) \Rightarrow \bigcup F\, y = F\, x\, y$.

Writing out the types is an instructive familiarization exercise for such functionals. An example for $^{\mathsf{T}}$ is $\mathsf{fam}\, \mathcal{F} \ni F \to \bigcap(\mathcal{D} \circ F) \to \mathcal{D}\, F \ni x \to \mathcal{R}(F\, x)$. In **def**-format,

$$\textbf{def}\ —^{\mathsf{T}} : \mathsf{fam}\, \mathcal{F} \ni F \to \bigcap(\mathcal{D} \circ F) \to \mathcal{D}\, F \ni x \to \mathcal{R}(F\, x)\ \textbf{with}\ F^{\mathsf{T}}\, y\, x = F\, x\, y.$$

Remark. The generic functionals constitute an interesting algebra of functions (point-free rules) and also point-wise rules. These are not stated here, except for one example: $f = \bigcup x : \mathcal{D}\, f \,.\, x \mapsto f\, x$ and, more remarkably, $f^{-} = \bigcup x : \mathcal{D}\, f \,.\, f\, x \mapsto x$, showing how well-matched the functionals designed via the stated principle are.

6. APPLICATION EXAMPLES: SECOND BATCH

6.1 Application to Mathematical and Programming Formalisms

Qua language design, we have shown how a very simple syntax makes common notations defect-free and generates new forms of expression whereby, contrary to common belief, formality increases freedom (e.g., $x \neq y \neq z$ to mean that x, y, z are distinct).

The formal rules endow predicate logic and discrete mathematics with the calculational style formerly found only in analysis and algebra.

The following application examples illustrate the ramifications of elastic and generic functionals. Occasional remarks indicate the practical origins of generic functionals.

6.1.1 *More Elastic Operator Design Examples and Ramifications*

6.1.1.1 *Proper Designs for $\sum$-Operators.* Operators over discrete structures are often *defined* recursively via an *empty rule*, a *one-point rule* and a *combining rule* [Bird 1998]. This also explains why we use τ for singleton tuples (the empty tuple already being ε).

For more general operators, these rules often emerge as *properties*, for instance $\exists\,\varepsilon \equiv 0$, $\exists(d \mapsto p) \equiv p$ and $P \mathbin{©} Q \Rightarrow (\exists(P \cup Q) \equiv \exists P \vee \exists Q)$, respectively.

We use this style in *defining* $\sum$, which is inherently more complex than $\exists$ or $\forall$, since $+$ is not idempotent like $\vee$ or $\wedge$. Here are the axioms, assuming any number-valued functions f and g with finite, nonintersecting domains, any d and any number c:

$$\begin{array}{lrcl} \textit{Empty rule}: & \sum \varepsilon & = & 0 \\ \textit{One-point rule}: & \sum (d \mapsto c) & = & c \\ \textit{Merge rule}: & f \cup g & = & \sum f + \sum g\,. \end{array} \tag{73}$$

Well-definedness is left as an exercise. Variadic $+$ is defined as in $x + y + z = \sum(x, y, z)$.

The ambiguities in common usage and software design errors as shown in Section 2.1 are avoided by making the vague common notation $\sum_{i=m}^{n} e$ precise as standing for $\sum i : m\,..\,n\,.\,e$, where $m\,..\,n$ is defined as in PASCAL [Jensen and Wirth 1978]. Aside: In calculations with intervals, it is advantageous to use m_n, including the lower but not the upper bound:

$$\begin{array}{lrcl} \text{PASCAL-like variant:} & m\,..\,n & = & \{i : \mathbb{Z} \mid m \le i \le n\} \\ \text{Preferable variant:} & m\_n & = & \{i : \mathbb{Z} \mid m \le i < n\} \end{array} \text{ for integer } m, n. \tag{74}$$

Formula (2) is obtained via algebraic properties of $\sum$, for example, *trading*: $\sum f_P = \sum(f \mathbin{\widehat{\cdot}} P)$.

A different design shows one way to correct (1) while preserving its right-hand side. Define a function transformer $—]_m^n$ by $f]_m^n = (m \le n)?(i : m\_n\,.\,f\,i) \mathbin{\dagger} (i : n\_m\,.\,-(f\,i))$ for $m, n : \mathbb{Z}^2$ and $f : \mathbb{Z} \to \mathbb{C}$. Then define the parameterized $\sum\!\!\!\!\!\!\sum : \mathbb{Z}^2 \to (\mathbb{Z} \to \mathbb{C}) \to \mathbb{C}$ with $\sum\!\!\!\!\!\!\sum_m^n f = \sum(f]_m^n)$. This ensures $\sum\!\!\!\!\!\!\sum_m^k f + \sum\!\!\!\!\!\!\sum_k^n f = \sum\!\!\!\!\!\!\sum_m^n f$ regardless of the order of the integers m, k, n. Note the analogy with $\int_a^c f + \int_c^b f = \int_a^b f$. This property is shared by a $\sum$ abstractor defined in Graham et al. [1994], based on expressions, but our variant is *functional* and developed independently to correct (1). Observe that, for *any* integers m, n, k,

$$\sum_{k}^{n} i : \mathbb{Z}\ .\ \sum_{i}^{m} j : \mathbb{Z}\ .\ 1 \;=\; \frac{(n-k)\cdot(2\cdot m - n - k + 1)}{2}.$$

Substituting $k := 0$ yields a salvaged version of (1) with the same right-hand side.

Another important advantage of definition (73) for $\sum$ is that the domain of the argument need not be numeric. Various *ad hoc* "shorthands" thereby obtain a formally correct version. For instance, in texts on electricity, Kirchhoff's current law at a circuit node K is often written sloppily as $\sum_K i = 0$. With our approach, writing i_K for the family of the currents flowing out of node K (i_{KL} for the current from K to L), one writes rigorously $\sum i_K = 0$, which illustrates the flexibility of domain modulation. Similar advantages in software specification and programming are mentioned later.

6.1.1.2 *Origin and Ramifications of the* $\bigtimes$*-Operator.* This interesting design arose from formalizing a convention for specifying a radio frequency filter characteristic f within a given tolerance [Carson 1990], for example, $m\,x \leq f\,x \leq M\,x$. Here is the approach [Boute 2003].

Let T be an interval-valued function, for example $T\,x = [m\,x, M\,x]$. A function f *meets tolerance* T iff $\mathcal{D}\,f = \mathcal{D}\,T$ and, for all x in the domain, $f\,x \in T\,x$. We generalize this by defining an operator $\bigtimes$ over *arbitrary set-valued functions* T as shown in (54).

The equivalent form $f \in \bigtimes T \equiv \mathcal{D}\,f = \mathcal{D}\,T \wedge \forall\, x : \mathcal{D}\,f \cap \mathcal{D}\,T\ .\ f\,x \in T\,x$ highlights the analogy with equality: $f = g \equiv \mathcal{D}\,f = \mathcal{D}\,g \wedge \forall\, x : \mathcal{D}\,f \cap \mathcal{D}\,g\ .\ f\,x = g\,x$, and hence $f = g \equiv f \in \bigtimes(\iota \circ g)$. In other words, the tolerance can be completely "tight".

By calculation, $X \times Y = \bigtimes(X, Y)$, which differs from the common Cartesian product only by the extra structure that tuples are functions. The extension is conservative. This explains our choice of the symbol $\bigtimes$ and the name *Functional Cartesian Product*. As expected, $\bigtimes$ is the elastic operator for defining variadic infix application of $\times$.

Expressions like $\bigtimes i : I\ .\ A_i$ capture the so-called *dependent type* or *product* [Tennent 1991], usually written $\prod_{i \in I} A_i$. However, the latter is just an ad hoc abstractor and $\bigtimes$ is a genuine functional, with algebraic properties not shared by the conventional forms and for which we have not yet seen counterparts elsewhere, such as the *inverse* $\bigtimes^-$.

The axiom of choice is equivalent to $\bigtimes T \neq \emptyset \equiv \forall\, x : \mathcal{D}\,T\ .\ T\,x \neq \emptyset$, which also characterizes $\mathsf{Bdom}\bigtimes$. If $\bigtimes T \neq \emptyset$, then $\bigtimes^-(\bigtimes T) = T$. For $\times$ this implies that, for nonempty sets, $\bigtimes^-(X \times Y) = X, Y$. Especially interesting is an explicit formula for the inverse: one can show calculationally that, for any nonempty S in the range of $\bigtimes$,

$$\bigtimes^- S = x : \bigcap(f : S\ .\ \mathcal{D}\,f)\ .\ \{f\,x \mid f : S\}. \tag{75}$$

With $\bigtimes$, we can formally express intricate conditions on arguments of a functional in its type expression (rather than separately in the surrounding text) as announced in Section 4.2.1. An illustration is the elastic lim-operator from (46):

$$\begin{aligned}&\textbf{def}\ \mathsf{lim}\ : (\mathbb{R} \nrightarrow \mathbb{R}) \ni f \to \{a : \mathsf{Ad}\,(\mathcal{D}\,f) \mid \exists\, b : \mathbb{R}\ .\ b\ \mathsf{islim}_f\ a\} \to \mathbb{R}\\ &\textbf{with}\ \forall\, f : \mathcal{D}\,\mathsf{lim}\ .\ \forall\, a : \mathcal{D}(\mathsf{lim}\ f)\ .\ (\mathsf{lim}\ f\ a)\ \mathsf{islim}_f\ a.\end{aligned}$$

Such cascaded conditions are even more apparent in the *Newton quotient functional*

$$\textbf{def}\ Q : FD \ni f \to \mathcal{D}\,f \ni x \to \mathcal{D}\,f \setminus \iota\,x \to \mathbb{R}\ \textbf{with}\ Q\ f\ x\ y = \frac{f\,y - f\,x}{y - x},$$

where *FD* is the set of real-valued functions whose domain is an *interval* [Lang 1983] of more than one point. This is used as an auxiliary function in defining the *derivation* operator with image definition $\mathrm{D}\,f\,x = \mathsf{lim}(Q\ f\ x)\,x$ and type definition left as an exercise.

6.1.1.3 *Variant: An Operator for Polymorphism.* Clearly, $\bigtimes$ is directly suitable for parameterized (Church-style) polymorphism, details being given in

[Boute 2003]. For completeness, we mention the *function type merge* ($\bigotimes$), designed by van den Beuken [1997] to express overloading and parameterless (Curry-style) polymorphism, and also discussed in [Boute 2003]: for any family G of function types, $\bigotimes G = \{\bigcup F \mid F : \times G \wedge \textcircled{c}\ F\}$.

6.1.2 *Unifying Structures as Functions in Mathematics and Programming.* Tuples, lists sequences and so on are ubiquitous structures and benefit most from being defined as functions. This intuitively evident design decision is surprisingly rare in the literature, where said structures are handled as entirely or subtly distinct from functions [Gierz et al. 1980; Lang 1983; Roberts and Mullis 1987; Wechler 1987]. The few exceptions [Halmos and Givant 1998] concern homogeneous cases only and do not exploit the function properties, thereby missing the advantages.

First, defining structures as functions bestows all function properties and generic functionals. Conversely, tuples guide the design of many elastic operators, including generic ones such as transposition. Moreover, all calculations and theorems about these structures as functions remain applicable even in formalisms and programming languages *not* viewing them as functions.

6.1.2.1 *Sequences.* We use this term to encompass tuples, arrays, lists and similar structures.

A *sequence* is any function with domain $\square\, n$ for some $n : \mathbb{N}'$. In this discussion, x and y stand for sequences. The *length* operator $\#$ is defined by $\#\, x = n \equiv \mathcal{D}\, x = \square\, n$.

Arrays of length n over a set A are functions of type A^n. For *lists*, $A^* = \bigcup\, n : \mathbb{N}\ .\ A^n$ (finite) and $A^\infty = \mathbb{N} \to A$ (infinite). We also define $A^\omega = A^* \cup A^\infty$. A *tuple* has type $\times S$ for some sequence S of nonempty sets. Clearly, $\times S \subseteq (\bigcup S)^{\#\, S} \subseteq (\bigcup S)^*$.

The usual recursive formulations and inductive arguments from the literature [Bird 1998] carry over in the functional formulation. For instance, as in [Boute 1991], we define the operators *prefixing* $(- \succ -)$ and *concatenation* $(- +\!\!+ -)$ for any e and sequences x and y by

$$e \succ x \ = \ i : \square\, (\#\, x + 1)\ .\ (i = 0)\, ?\, e \mid x\, (i - 1) \tag{76}$$

$$x +\!\!+ y \ = \ i : \square\, (\#\, x + \#\, y)\ .\ (i < \#\, x)\, ?\, x\, i \mid y\, (i - \#\, x). \tag{77}$$

The formulas $\varepsilon +\!\!+ y = y$ and $(e \succ x) +\!\!+ y = e \succ (x +\!\!+ y)$ can be seen as either theorems derived from (77) or a recursive definition replacing (77). Elastic operators applied to lists enjoy properties like $\sum (a \succ x) = a + \sum x$ (theorems, not definitions). Structural induction on A^* has the usual formulation but is again a theorem:

$$P\, \varepsilon \wedge \forall (x : A^*\ .\ P\, x \Rightarrow \forall a : A\ .\ P\, (a \succ x)) \ \Rightarrow\ \forall x : A^*\ .\ P\, x. \tag{78}$$

For infinite lists, one can similarly reformulate co-induction, but in many cases just induction on the index suffices because $A^\infty = \mathbb{N} \to A$.

To further illustrate the versatility of $\times$ for typing sequences: if S and T are sequences of sets, then any $x : \times S$ and $y : \times T$ satisfy $x +\!\!+ y \in \times (S +\!\!+ T)$.

6.1.2.2 *Records and Other Structures.* *Records* in the sense of PASCAL [Jensen and Wirth 1978] are important in programming. One functional approach, similar to projection in category theory, uses *selector functions* corresponding to field labels, and the records are arguments. This is used in Haskell [Hudak et al. 1999], and was also explored earlier in an other context [Boute 1982]. However, it does not view records as functions, and has a rather heterogeneous flavor.

With $\times$ we can define the records themselves as *functions* whose domain is a set of field labels defined as an *enumeration type* [Jensen and Wirth 1978], for instance, {name, age} in

$$Person := \times(\texttt{name} \mapsto \mathbb{A}^* \cup \texttt{age} \mapsto \mathbb{N}).$$

This defines a function type such that $person : Person$ satisfies $person\ \texttt{name} \in \mathbb{A}^*$ and $person\ \texttt{age} \in \mathbb{N}$. Obviously, by defining $\mathsf{Record}\ F = \times(\bigcup F)$, one can also define $Person := \mathsf{Record}\ (\texttt{name} \mapsto \mathbb{A}^*, \texttt{age} \mapsto \mathbb{N})$, which is more suggestive.

We define *trees* as functions whose domains are *branching structures*, that is, sets of sequences describing the path from the root to a leaf in the obvious way (for any branch labeling). For instance, for a binary tree, the branching structure is a subset of $\mathbb{B}^*$.

Data structures defined or modelled as functions inherit all elastic and generic functionals, which facilitates reasoning about programs using them. Given a data structure $s : D \to \mathbb{Z}$ (sequence, a tree etc. depending on the structure of D) we can simply write, for instance, $\sum s$ for the sum of the elements if they are numbers.

6.1.3 *Applications of Generic Functionals in Mathematics and Programming*

6.1.3.1 *Mathematical Software.* For good reasons, mathematical software often tries to accommodate common usage, but thereby inherits deficient parts as well. Typical are floating pieces of syntax and operator arguments that are not expressions but ad hoc constructs merging poor conventions from discrete mathematics with imperative constructs from the implementation. For instance, *iterators* introduce a *range variable* and specify step size and boundaries, such as `i=1..n` in `sum(sum(1,j=i..m),i=1..n)`.

Using domain modulation instead would yield a definite improvement in existing mathematical software (MATLAB, Mathcad, Maple, Mathematica etc.). The resulting expressions are conceptually simple, clear, compact, and have precise calculation rules.

More generally, our approach markedly simplifies languages by using few constructs.

6.1.3.2 *Obviating Ellipsis.* An entrenched mathematical convention is ellipsis (Section 2.1). It is a clumsy way of expression and hampers designing precise rules. “Continuous” mathematics has the fortune of lacking this “feature”. Elastic and generic functionals, together with sequences as functions, make complete cleanup a routine matter.

Lengthy expressions like $f\,x_0, f\,x_1, \ldots, f\,x_{n-1}$ are replaced by $\overline{f}\,x$ or $\overline{f}\,x_{<n}$ as needed. For $f\,x_0, f\,x_1, \ldots, f\,x_{m-1}, e, f\,x_{m+1}, \ldots, f\,x_{n-1}$ one writes $\overline{f}\,x \oslash (m \mapsto e)$ (or defines an operator on this basis). All redundancy that has no redeeming value is gone, which also improves clarity. Such use of generic functionals is not limited to numerical domains.

Furthermore, we define a sequence constructor $\cdots$ with, for integer m, n,

$$m \cdots n = i : \square\, |n-m|\,.\,(m \le n)\,?\,(m+i) \nmid (m-1-i) \tag{79}$$

This replaces $f_m, f_{m+1}, \ldots, f_{n-1}$ by $f \circ (m \cdots n)$, assuming $m \le n$.

More generally, to fully appreciate the use of generic functionals in obviating ellipsis, it suffices to apply these ideas to a few randomly selected papers.

Unifying functions, tuples, lists and so on results in useful algebraic properties. Noteworthy are composition, for example, $(3,4,5,6) \circ (1,0,2) = 4,3,5$, and inverses, for example, $(3,3,7)^{-}\,7 = 2$. An application of list inverses in formal semantics is shown later.

6.1.3.3 *Applications in Programming.* Generic functionals subsume many special-purpose functions or conventions and generalize them to arbitrary functions. Examples from functional programming, mainly from Haskell [Bird 1998; Hudak et al. 1999], are written in `typewriter font`.

Composition ($\circ$ or $^{-}$) subsumes the usual *map* operator, for example, `map f [a, b, c] = [f a, f b, f c]` is subsumed by $\overline{f}\,(a,b,c) = f\,a, f\,b, f\,c$, assuming $\{a,b,c\} \subseteq \mathcal{D}\,f$.

Transposition ($^{\mathsf{T}}$) is the dual of composition in various ways, for example, the distributivity laws (provided $y \in \mathcal{D}\,f \cap \mathcal{D}\,g$ and $\{x, y\} \subseteq \mathcal{D}\,f$, respectively)

$$(f,g)^{\mathsf{T}}\,y = f\,y, g\,y \quad \text{and} \quad f \circ (x,y) = f\,x, f\,y. \tag{80}$$

In the discrete domain of functional programming, this subsumes the `zip` operator: `zip[[a,b,c],[a′,b′,c′]] = [[a,a′],[b,b′],[c,c′]]` (up to Currying), if lists are modelled as functions. Indeed, $((a,b,c),(a',b',c'))^{\mathsf{T}} = (a,a'),(b,b'),(c,c')$.

More generally, given $f : A \to B \to C$ (nonempty A, B, C), then $f^{\mathsf{T}} \in B \to A \to C$ and $(f^{\mathsf{T}})^{\mathsf{T}} = f$, a useful algebraic law for functional programs of the given type.

Direct extension ($\bar{}$, $\hat{}$ etc.) more recently also made its appearance in programming, but without generic design. In Dijkstra and Scholten's [1990] program semantics, operators are inherently extended to structures, for example, for arithmetic $+$, as expressed by $(x+y).n = x.n + y.n$. This even includes equality: if x and y are structures, then $x = y$ does not express equality of x and y but a function with $(x=y).n \equiv x.n = y.n$, that is, point-by-point equality. *Polymorphism* in LabVIEW [Bishop 2001] is a similar implicit extension.

Implicit extension is reasonable in special areas but too rigid for general practice. An explicit operator offers more flexibility and generalization via our design principle. Hence, we write $f \mathrel{\widehat{=}} g$ for point-by-point equality, and $f = g \equiv \mathcal{D}\,f = \mathcal{D}\,g \wedge \forall (f \mathrel{\widehat{=}} g)$ for "equality everywhere" (function equality). Explicit extension is evidently also useful for describing the semantics of languages where extension is implicit [Bishop 2001].

Function inverse has an interesting application in modelling *pattern matching* as in functional languages, for example, $f\,(a \succ\!\!- x) = h\,(a,\, f\,x)$ in recursive definitions over lists. This can be seen as a special form of more general implicit definition, allowing not just constructors but also other functions. Given the more general form $f\,(g\,(a,x)) = h\,(a,\, f\,x)$, application of f to an actual parameter $y : \mathsf{Bran}\,g$ satisfies $f\,y = h\,(g^-\,y\,0,\, f\,(g^-\,y\,1))$. In the list example, $\mathsf{Bran}(\succ\!\!-) = A^+$, so $(\succ\!\!-)^-\,(a \succ\!\!- x)\,0 = a$ and $(\succ\!\!-)^-\,(a \succ\!\!- x)\,1 = x$.

The *funcart product* $\times$ offers high flexibility to express types in programming languages. This is most relevant for languages that do not treat types as first-class objects and hence lack flexibility by themselves. Types often have their own sublanguage with conventions that, in an orthogonal language, would be considered an abuse of notation.

Typical in many functional languages is expressing the type of a pair `(x,y)` as a pair `(X,Y)`, meant to express the Cartesian product $\mathtt{X} \times \mathtt{Y}$. Similarly, whereas `[a]` stands for $\tau\,\mathtt{a}$, the similar `[A]` stands for $\mathtt{A}^*$. Some design decisions go back to certain conventions in category theory, which luckily are easy to improve (not discussed here).

In such languages, properties like $x +\!\!+ y \in \bigtimes(S +\!\!+ T)$ are hard to express, hence using Funmath as a metalanguage is most appropriate. Educationally, distinguishing (A, B) from $A \times B$ right from the start avoids later confusion and creates a conceptual framework in which particularities of programming languages can be clearly explained.

6.2 Generic Functionals vs. Domain-Specific Language Concepts

Paradoxically, generic functionals prove directly usable in areas where most language designs resort to domain-specific concepts. Here are some "snapshots" from Boute [2003].

6.2.1 *Hardware Description Languages (HDLs).* Hardware description is the first topic since it provided the original motivation for our operator design, and the chosen example yields an interesting fixpoint equation.

Let *system behaviors* be modelled as functionals from *signals* (functions of continuous or discrete time) to signals. By design, the generic functionals capture system description constructs: cascade connection by $f \circ g$, replication by $\overline{f}\,(s, s') = f\,s,\, f\,s'$, input sharing by $(f, g)^\mathsf{T} s = f\,s, g\,s$, signal pairing by $(s, s')^\mathsf{T}\,t = s\,t, s'\,t$ and so on.

Hence, generic functionals can fulfil the same purpose as the various special-purpose operators in Hudak et al. [2003] for combining subsystems in reactive and robotics applications.

Likewise, they are very convenient in calculational transformation, for example, from behaviors (specifications) to structures (realizations) by eliminating the time variable.

An example is the recursive definition (with given set A, $a : A$ and $g : A \to A$)

$$\textbf{def}\ f : \mathbb{N} \to A\ \textbf{with}\ f\,n = (n = 0)\,?\,a \nmid g\,(f\,(n-1)). \tag{81}$$

The image definition can be transformed calculationally as follows. For any n in $\mathbb{N}$,

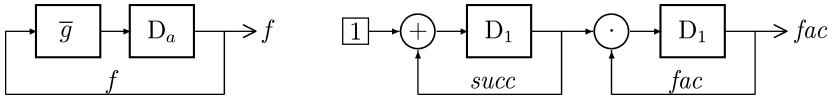
$$\begin{aligned} f\,n &= \langle \text{Defin. } f(81), \circ \rangle \quad (n=0)\,?\,a \mid (g \circ f)(n-1) \\ &= \langle \text{Defin. D below} \rangle \quad \mathrm{D}_a\,(g \circ f)\,n \\ &= \langle \text{Definition} =, \circ \rangle \quad (\mathrm{D}_a \circ \overline{g}\,)\,f\,n, \end{aligned}$$

where $\mathrm{D}_{\_\_} : A \to (\mathbb{N} \to A) \to (\mathbb{N} \to A)$ is defined by $\mathrm{D}_a\,x\,n = (n=0)\,?\,a \mid x\,(n-1)$.

By function equality, this yields the fixpoint equation $f = (\mathrm{D}_a \circ \overline{g}\,)\,f$, which has the same solution for f as the recursion equation (81). It also has a very concrete signal flow interpretation with $\mathbb{N}$ as the time domain. D is then interpreted as the behavior of a clocked memory cell containing one element of type A, with given initial contents.

Similarly, given $fac : \mathbb{N} \to \mathbb{N}$ with $fac\,n = (n = 0)\,?\,1 \mid n \cdot fac\,(n-1))$, the image definition can be transformed into $fac = \mathrm{D}_1\,(succ \mathbin{\hat{\cdot}} fac\,)$ where $succ = \mathrm{D}_1\,(1 \mathbin{\vec{+}} succ)$.

Block diagrams for systems realizing the three fixpoint equations are



Such realizations are directly implementable in signal flow languages such as LabVIEW [Bishop 2001], a graphical language for instrumentation often used in laboratories and plants.

From a programming point of view, recursion has been transformed into iteration.

In the domain of HDLs such as VHDL [IEEE 1994] the variety of constructs for system description in general and for detailed issues such as array shifting, subarray selection etc. is similarly captured by the generic functionals. As a result, everything one can express in VHDL can be expressed at least as conveniently in our formalism [Boute 1993b].

This was found useful in describing formal semantics of system description languages ranging from textual ones such as VHDL to graphical ones such as LabVIEW [Boute 2003]. As the next item, we shall consider programming language semantics.

6.2.2 *Programming Language Semantics.* Most approaches to formal semantics uses special-purpose conventions and operators [Meyer 1991; Tennent 1991]. We show how the various operators of Section 5 are directly usable instead.

For example, to describe *abstract syntax*, Metanot [Meyer 1991] is subsumed by elastic and generic functionals. *Repetition constructs* are covered by the *-operator, for example,

$$\textbf{def}\ \mathit{Declist} := \mathit{Declaration}^*.$$

Aggregate constructs are covered by $\times$ or its `Record` variant:

$$\begin{aligned} &\textbf{def}\ \mathit{Program} := \mathsf{Record}\,(\mathsf{decl} \mapsto \mathit{Declist}, \mathsf{body} \mapsto \mathit{Instruction}) \\ &\textbf{def}\ \mathit{Declaration} := \mathsf{Record}\,(\mathsf{var} \mapsto \mathit{Variable}, \mathsf{typ} \mapsto \mathit{Type}) \\ &\textbf{def}\ \mathit{Assignment} := \mathsf{Record}\,(\mathsf{target} \mapsto \mathit{Variable}^*, \mathsf{source} \mapsto \mathit{Expression}^*). \end{aligned}$$

To express *choice constructs* requiring disjoint union, we define an elastic operator | by $|\ T = \bigcup x : \mathcal{D}\, T\ .\ \{x \mapsto y \mid y : T\, x\}$ for any set-valued function T. The expression $A \mid B \mid C$ familiar from BNF is now just the variadic infix application for $|\ (A, B, C)$, for example,

def *Instruction* := *Skip* | *Assignment* | *Compound* | *Conditional* | *Loop*.

Labels thus introduced are numbers, but more enlightening labels can be given, as in $|\ (\mathsf{skip} \mapsto \mathit{Skip} \cup \mathsf{asgn} \mapsto \mathit{Assignment}$ etc.). In fact, disjoint union is useful for this purpose only, since syntactic domains are mostly disjoint and regular union suffices.

Abstract syntax trees are then Curried versions of branching structures. Semantic functions can be used for mapping abstract to concrete syntax.

To define *semantic functions* and *calculate* with them, generic functionals are ideal. For instance, in a functional recast of Meyer's [1991] formulation, static semantics for *validity* of declaration lists (no double declarations) and the variable inventory are

$$\textbf{def}\ \mathit{Valdcl} : \mathit{Declist} \to \mathbb{B}\ \ \textbf{with}\ \mathit{Valdcl}\ dl = \mathsf{inj}\,(dl^{\mathsf{T}}\mathsf{var})$$
$$\textbf{def}\ \mathit{Vars} : \mathit{Declist} \to \mathcal{P}\,\mathit{Variable}\ \textbf{with}\ \mathit{Vars}\ dl = \mathcal{R}\,(dl^{\mathsf{T}}\mathsf{var}).$$

The (static) semantic function *Tmap* derives from a valid declaration list a *type map* [Meyer 1991] from declared variables to their types.

$$\textbf{def}\ \mathit{Tmap} : \mathit{Declist}_{\mathit{Valdcl}} \ni dl \to \mathit{Vars}\ dl \to \mathit{Type}\ \textbf{with}$$
$$\mathit{Tmap}\, dl = (dl^{\mathsf{T}}\mathsf{typ}) \circ (dl^{\mathsf{T}}\mathsf{var})^{-}.$$

A style variant: defining $\mathit{Vars}\ dl = \mathsf{Bran}\,(dl^{\mathsf{T}}\mathsf{var})$ obviates filtering by *Valdcl*. Instantiating $f^{-} = \bigcup x : \mathcal{D}\, f\ .\ f\, x \mapsto x$ with $f := dl^{\mathsf{T}}\mathsf{var}$ is the central step in showing that $\mathit{Tmap}\, dl = \bigcup d : \mathcal{R}\, dl\ .\ d\ \mathsf{var} \mapsto d\ \mathsf{typ}$. This, and $i \in \mathcal{D}\, dl \Rightarrow \mathit{Tmap}\, dl\, (dl\, i\ \mathsf{var}) = dl\, i\ \mathsf{typ}$ for valid dl, yields alternative definitions, for example, for validation. Indeed, although no method can show that a formula reflects what is intended, one can obtain confidence by reflecting the intention through nontrivially different formulas via separate routes, and proving equivalence. This principle is crucial in validating formal specifications.

Dynamic semantics is similar in style, especially the use of generic functionals [Boute 2003].

In brief, using general operators we covered the essence of several dozens of pages from Meyer [1991] (using three kinds of domain-specific operators) within a few paragraphs.

Similar observations apply to other structures such as directories used on most computers, XML and relational databases [Boute 2003].

6.2.3 *Relational Databases in a Functional Formalism.* Database systems store information and present an interface to the user for retrieving it in the form of *virtual tables*. A *relational database* presents the tables as relations, which makes including column headings awkward. We define table rows as *records* in the functional way, with column headings as field names. This embeds databases in a wider framework with more general algebraic properties and using generic functionals.

For handling virtual tables in constructing queries, the main operations usually are *selection*, *projection* and *natural join* [Gries and Schneider 1993]. Since, in our formalism, tables are sets of records (functions), generic functionals directly provide all desired functionality [Boute 2003].

—The *selection* operator (σ) selects for any table $S : \mathcal{P}\, R$ of records of type R precisely those records satisfying a given predicate $P : R \to \mathbb{B}$. This is achieved by the set filtering operator in the sense that $\sigma\,(S, P) = S \downarrow P$.

—The *projection* operator (π) yields for any table S of records a subtable containing only the columns corresponding to a given set F of field names. This is just a variant of function domain restriction expressible by $\pi\,(S, F) = \{r \restriction F \mid r : S\}$.

—The *("natural") join* operator ($\bowtie$) combines tables S and T by uniting the domains of the elements (field name sets), but keeping only those records for which field names common to both tables have the same contents, that is, only *compatible* records are combined: $S \bowtie T = \{s \cup t \mid (s, t) : S \times T \wedge s \mathbin{©} t\}$. This is precisely the earlier *function type merge* in the sense that $S \bowtie T = S \otimes T = \bigotimes(S, T)$.

The algebraic properties of generic functionals allow proving that, in contrast with function merge ($\cup$), function type merge ($\otimes$) and hence join ($\bowtie$) is associative.

6.3 Examples from Informal Statements to Formal Calculation

These examples are chosen to highlight another issue in parallel, namely how binary algebra as arithmetic combines with elastic operators in more general contexts.

6.3.1 *Formalizing and Solving Parameterized Word Problems.* As shown by Gries and Schneider [1993], word problems are good exercises in formalizing specifications. A valuable bonus is demystifying certain self-referential statements by viewing them as (systems of) equations that may have zero or multiple solutions. For instance, "This statement is false." is formalized as $x \equiv \neg\, x$, an equation with no solution.

Such exercises are not a luxury: borrowing some simple examples used by Johnson-Laird [2000], [http://webscript.princeton.edu/~psych/PsychSite/fac_phil.html] for investigating fallacious logic reasoning by humans, we found that difficulties were not only experienced by novice computer science students [Almstrum 1996], but also caused serious errors by those who had an earlier course on formal logic elsewhere.

Word problems with a fixed number of statements fall under proposition calculus. However, the problem: "How many of the following statements list are true?

1. Exactly 1 of these statements is false.
...
10. Exactly 10 of these statements are false."

benefits from *parameterizing* it in the number of statements, say n, and using predicates. Let us define $P : 0\,..\,n \to \mathbb{B}$ with $P\,i \equiv$ "exactly i of the n given statements are false". Formalizing this via the arguments in [Boute 2004] yields the specification

$$\textbf{spec}\ P : 0\,..\,n \to \mathbb{B}\ \ \textbf{with}\ P\,i \equiv \sum(i : 1\,..\,n\,.\,\neg P\,i) = i.$$

Solving this for P leads to the requirement $n \geq 2$ and a unique solution, $P\,i \equiv i = n - 1$, hence **spec** may be replaced by **def**. The calculation details can be found in [Boute 2004].

Replacing "false" by "true" in every statement yields a problem with two solutions.

6.3.2 *Formal Specification and Program Correctness.* The chosen example is *sorting*. We found it also useful in various courses to illustrate some essential but often-neglected points, which are also recalled below.

6.3.2.1 *Abstract Specification.* Let A be a set with total order $\sqsubseteq$. Sorting means that the result is ordered and has the same content. First point: Students asked to formalize "sorting" often forget the latter requirement as being "evident". We specify

$$\textbf{spec}\ \mathit{sort} : A^* \to A^*\ \textbf{with}\ \mathit{ndesc}\,(\mathit{sort}\,x) \wedge (\mathit{sort}\,x)\,\mathit{samecontents}\,x,$$

postponing auxiliary functions. Second point: Often existence and uniqueness is assumed tacitly, especially when "specifying" sorting in a programming language.

The predicate *ndesc* reflects the usual choice, viz., "nondescending" or, loosely, "ascending". Modulo seeing lists as functions, definitions in the literature amount to:

$$\textbf{def}\ \mathit{ndesc} : A^* \to \mathbb{B}\ \ \textbf{with}\ \mathit{ndesc}\,x \equiv \forall(i, j) : (\mathcal{D}\,x)^2\,.\,i \leq j \Rightarrow x\,i \sqsubseteq x\,j.$$

For *samecontents*, most formulations amount to $y\ \mathit{samecontents}\,x \equiv y\ \mathsf{perm}\ x$ using a *permutation* relation, say, $g\ \mathsf{perm}\ f \equiv \exists\,\pi : \mathcal{D}\,f \to \mathcal{D}\,g\,.\,(\pi\ \mathsf{bij}\,\mathcal{D}\,g) \wedge f = g \circ \pi$.

We prefer a less circuitous definition [Boute 1993a], expressing contents by *functions*. An appropriate data structure to reflect contents (ignoring order) is a *multiset* or *bag*, also used to specify sorting in Cohen [1990]. However, we define bags as functions: $\mathsf{bag}\,A = A \to \mathbb{N}$ with the idea that, for $b : \mathsf{bag}\,A$ and $a : A$, the number of a's in b is $b\,a$. In this setting, fuzzy predicates, ordinary predicates and bags over A are all defined as "degree of membership" functions of type $A \to B$ where B is $[0, 1]$, $\{0, 1\}$ and $\mathbb{N}$ respectively.

With this preamble, we define an *inventory* operator that maps functions to bags:

$$\textbf{def}\ \S : (\mathsf{fam}\,A)_{\mathsf{fin}} \to \mathsf{bag}\,A\ \textbf{with}\ \S\,f\,a = \sum x : \mathcal{D}\,f\,.\,a = f\,x,$$

again based on binary algebra. Defining $y\ \mathit{samecontents}\,x \equiv \S\,y = \S\,x$ allows rewrite the specification as **spec** $\mathit{sort} : A^* \to A^*$ **with** $\mathit{ndesc}\,(\mathit{sort}\,x) \wedge \S\,(\mathit{sort}\,x) = \S\,x$. Proving existence and uniqueness (so **spec** may become **def**) is not as evident as it may look.

This specification is declarative but, recast as a relation $-sorted-$: $(A^*)^2 \to \mathbb{B}$ with $y\ sorted\ x \equiv ndesc\ y \wedge \S\ y = \S\ x$ to be used for testing, it is directly implementable.

6.3.2.2 *Implementation.* A typical (functional) program implementation is

def $qsort : A^* \to A^*$ **with**
$qsort\ \varepsilon = \varepsilon \wedge$
$qsort\ (a \succ x) = qsort\ u \prec a \succ qsort\ v$ **where** $u, v := split\ a\ x$
def $split : A \to A^* \to A^* \times A^*$ **with**
$split\ a\ \varepsilon = \varepsilon, \varepsilon \wedge$
$split\ a\ (b \succ x) = (a \sqsubseteq b)\,?\,(u, b \succ v) \nmid (b \succ u, v)$ **where** $u, v := split\ a\ x$

6.3.2.3 *Verification.* The proof obligations are clearly $ndesc\ (qsort\ x)$ and $\S\ (qsort\ x) = \S\ x$.

For the proof, we give an outline only; more details are found in [Boute 1993a]. Based on a simple problem analysis [Boute 2004], we introduce functions *le* and *ge*, both of type $A \times A^* \to \mathbb{B}$, with $a\ le\ x \equiv \forall i : \mathcal{D}\ x\ .\ a \sqsubseteq x\ i$ and $a\ ge\ x \equiv \forall i : \mathcal{D}\ x\ .\ x\ i \sqsubseteq a$.

Properties most relevant here are expressed by two lemmata: for any x and y in A^* and a in A, and letting $u, v := split\ a\ x$, we can show:

	Split lemma	Concatenation lemma
(a)	$\S\ u \mathbin{\widehat{+}} \S\ v = \S\ x$	$\S\ (x +\!\!+ y) = \S\ x \mathbin{\widehat{+}} \S\ y$
(b)	$a\ ge\ u \wedge a\ le\ v$	$ndesc\ (x +\!\!+ \tau\ a +\!\!+ y) \equiv ndesc\ x \wedge ndesc\ y \wedge a\ ge\ x \wedge a\ le\ y$

Combining $\S\ (x +\!\!+ y) = \S\ x \mathbin{\widehat{+}} \S\ y$ with $\S\ \varepsilon = A^\bullet 0$ and $\S\ (\tau\ a) = (= a)\rceil A$ show § to be a *list homomorphism* as defined in Bird [1998]. The properties $ndesc\ \varepsilon \equiv 1$ and $ndesc\ (a \succ x) \equiv a\ le\ x \wedge ndesc\ x$ and the mixed property $\S\ x = \S\ y \Rightarrow ge\ x = ge\ y \wedge le\ x = le\ y$ are the ingredients to make the proof of $ndesc\ (qsort\ x)$ and $\S\ (qsort\ x) = \S\ x$ a simple exercise.

6.4 A More Extended Case Study: A Minitheory of Recursion

Weaknesses in typical treatments of program semantics are: (a) specialized conventions that are not useful for other topics, (b) proofs only in the usual informal style (having no formal rules for the metalanguage, or assuming the reader has rules from elsewhere).

This case study shows how our general formalism allows deriving a simple theory of recursion with more complete proofs, yet still more compact, than found elsewhere. It also covers the link between denotational and axiomatic semantics, invariants and bound functions in loops but also in recursion—where they are a neglected topic—and a complete example. The author found this exercise very helpful in answering many questions essential to understanding, yet not addressed in available texts.

It also demonstrates how avoiding "partial functions" by precise domain definitions imposes no burden but, to the contrary, is convenient in formal derivations.

Table V. Illustration of Data structures derived from the declaration list

Item	Example
Declaration	`var k : int, b : bool`
State space	$S = \bigtimes(\mathtt{k} \mapsto \mathbb{Z} \cup \mathtt{b} \mapsto \mathbb{B})$
Tuple representation	$T = \bigtimes(0 \mapsto \mathbb{Z} \cup 1 \mapsto \mathbb{B}) = \mathbb{Z} \times \mathbb{B}$
State shorthand	$\langle\text{—}\rangle : T \to S$ with $\langle k, b\rangle = \mathtt{k} \mapsto k \cup \mathtt{b} \mapsto b$

6.4.1 *Conventions Regarding Formal Semantics and State Space Refinement.* In every simple imperative language, one can identify syntactic categories E for *expressions* (including V, the *variables* and B, the Boolean expressions) and C for *commands*.

(a) *Denotational semantics* usually defines a domain D, *state space* $S := V \to D$, and *meaning functions* $\mathcal{E} : E \to S \to D$ for expressions, $\mathcal{C} : C \to S \to S$ for commands.

For practical reasons, we refine the state space, assuming all variables typed. The data structures needed are derived from the declaration list, as illustrated in Table V. Mappings needed are found in Section 6.2.2. Using $\langle\text{—}\rangle$ removes much clumsiness from denotational semantics in reasoning about programs. Ignored by theory, switching between state formats is important in practice, and supported by generic functionals.

(b) *Axiomatic semantics* is usually expressed by Hoare triples $[a]c[p]$ for command c, precondition a ("ante") and postcondition p, both in some assertion language A.

The usual treatments vary from viewing it as a topic in metamathematics, with emphasis on nomenclature, consistency and completeness [Loeckx and Sieber 1984; Winskel 1993], to a calculus for developing programs, just stating the rules and then proceeding to applications [Cohen 1990; Gries and Schneider 1993].

We adopt a middle ground. Omitting some technicalities, we embed A in E and let

$$[a]c[p] \quad \text{stand for} \quad \forall s : S \,.\, \mathcal{E}\, a\, s \Rightarrow \mathcal{Q}\, c\, s \wedge \mathcal{E}\, p\, (\mathcal{C}\, c\, s), \tag{82}$$

where $\mathcal{Q} : C \to S \to \mathbb{B}$ is the *termination meaning function* to cover total correctness.

If one feels (82) too model-biased for axiomatic-style proofs, consider that the usual "axioms" can be derived as theorems, but their form and actual use remain identical.

Only the most interesting cases are exemplified: iteration for imperative languages and related forms of recursion for functional languages. We model iteration by

$$\mathcal{C}\,[\![\mathtt{while}\, b\, \mathtt{do}\, c\, \mathtt{od}]\!]\, s \;=\; \mathcal{E}\, b\, s\, ?\, \mathcal{C}\,[\![\mathtt{while}\, b\, \mathtt{do}\, c\, \mathtt{od}]\!]\, (\mathcal{C}\, c\, s) \mathbin{\mathbf{|}} s \tag{83}$$

and use its abstract form $f\, x = B\, x\, ?\, f\, (g\, x) \mathbin{\mathbf{|}} h\, x$ as an example of a recursion equation. Calculation will yield (a) a termination condition, (b) an explicit solution, (c) recursion invariants and bound functions and, via (83), similar results for iteration.

6.4.2 *Warming Up: Recursion and Iteration with Explicit Bounds.* For example (81), we derived a fixpoint equation $f = (D_a \circ \overline{g})\, f$. Given that $\mathcal{D}\, f = \mathbb{N}$ and $g \in A \to A$, it has a unique solution, namely $f\, n = g^n\, a$ (proof by induction).

Parameterizing (81) as **def** $f\_ : A \to \mathbb{N} \to A$ **with** $f_a\, n = (n = 0)\,?\, a \;\mathbin{|}\; g\,(f_a\,(n-1))$ allows expressing the *syphoning property*: $\forall\, a : A\,.\, \forall\, n : \mathbb{N}\;.\; f_{g\,a}\, n = f_a\,(n+1)$, proven by induction. This yields an imperative program: declaring `var a : A, n : nat`, the loop

$$loop := [\![\texttt{while n /= 0 do a, n := g a, n - 1 od}]\!].$$

leads by (83) to the equation $\mathcal{C}\, loop\, \langle a, n\rangle = (n \neq 0)\,?\, \mathcal{C}\, loop\, \langle g\, a, n-1\rangle \;\mathbin{|}\; \langle a, n\rangle$ and a

THEOREM : $\mathcal{C}\, loop\, \langle a, n\rangle = \langle f_a\, n, 0\rangle$,

proven by induction using the syphoning property. Hence, $\mathcal{C}\, loop\, \langle a, n\rangle\, \mathtt{a} = f_a\, n = g^n\, a$.

Such direct or *ingenuous* [Loeckx and Sieber 1984] proofs are simple but repetitive. This is avoided by encapsulating them once and for all in axiomatic semantics, as shown in Section 6.4.5.

6.4.3 *A Generic Recursion Equation and Its Solution*

6.4.3.1 *A Generic Recursion Equation.* As announced, we consider the equation

$$f\, x = B\, x\,?\, f\,(g\, x) \;\mathbin{|}\; h\, x, \tag{84}$$

where B is a given predicate and g and h given functions. The unknown is function f, with the added conditions that the definition of $\mathcal{D}\, f$ must ensure termination, and out-of-domain applications are avoided in (84). The latter condition is

$$x \in \mathcal{D}\, f \Rightarrow x \in \mathcal{D}\, B \wedge (B\, x \Rightarrow x \in \mathcal{D}\, g \wedge g\, x \in \mathcal{D}\, f) \wedge (\neg B\, x \Rightarrow x \in \mathcal{D}\, h). \tag{85}$$

Similar conditions are imposed on B, g and h, recursively propagating the conditions via the function domains. Assuming the domain of interest is X, we can write

$$\mathcal{D}\, f \subseteq X \quad X \subseteq \mathcal{D}\, B \quad X_B \subseteq \mathcal{D}\, g \quad \mathcal{R}\,(g \restriction X_B) \subseteq X \quad X \overline{\restriction}_B \subseteq \mathcal{D}\, h \tag{86}$$

in *partial* fulfilment of (85). Specifying $\mathcal{R}\,(g \restriction X_B) \subseteq X$ instead of $\mathcal{R}\,(g \restriction X_B) \subseteq \mathcal{D}\, f$ separates concerns ($\mathcal{D}\, f$ is part of the unknown) but requires verifying (85) afterwards.

6.4.3.2 *Solving the Equation.* Generally f is not total on X, but instead of using "partial" functions (Section 3.2.2) and solving (84) for f in $X \nrightarrow \mathcal{R}\, h$, we shall determine $\mathcal{D}\, f$ as the subset of X to satisfy termination and (85), and an explicit image formula for f. The rules for conditionals [Boute 2000] allow writing (84) as

$$\forall\, x : \mathcal{D}\, f\;.\; (B\, x \Rightarrow f\, x = f\,(g\, x)) \wedge (\neg B\, x \Rightarrow f\, x = h\, x). \tag{87}$$

We understand "termination" in the mechanics of actual execution, and do not just "postulate" (as most treatments with partial functions do) that it is just what our mathematical formulas model. These mechanics mean unfolding $B\,x \Rightarrow f\,x = f\,(g\,x)$ as $B\,x \Rightarrow f\,x = B\,(g\,x)\,?\,f\,(g\,(g\,x)) \nmid h\,(g\,x)$. The resulting pattern shows that the elements of X that ensure termination are characterized by the *termination predicate*

$$\textbf{def}\ Q : X \to \mathbb{B}\ \ \textbf{with}\ Q\,x \equiv \exists\, n : \mathbb{N}\ .\ \neg B\,(g^n\,x); \tag{88}$$

hence, $\mathcal{D}\,f = X_Q$. The same pattern yields $f\,x = h\,(g^{\mu\,x}\,x)$, where $\mu\,x$ is the least natural number such that $\neg B\,(g^{\mu\,x}\,x)$, is the explicit image definition. So we have the solution, and readers not interested in further details may wish to skip to Section 6.4.4.

6.4.3.3 *Formal Verification.* To formally verify that the solution solves (84) and meets (86), we note that g^n is recursively defined by $g^0 = id$ (identity function) and $g^{n+1} = g \circ g^n$ with the generic $\circ$ (13). The desired least element is obtained indirectly via the g.l.b.

$$\textbf{def}\ \mu : X \to \mathbb{N}'\ \textbf{with}\ \mu\,x = \curlywedge\, n : \mathbb{N}\ \mid \neg B\,(g^n\,x). \tag{89}$$

Indeed, by the case study in Section 4.2.3, μ also yields the desired least element since

$$Q\,x \equiv \mu\,x \in \mathbb{N}\ \wedge \neg B\,(g^{\mu\,x}) \wedge \forall\, n : \mathbb{N}\ .\ n < \mu\,x \Rightarrow B\,(g^n\,x), \tag{90}$$

whereas $\neg Q\,x \equiv \mu\,x = \infty$. This yields the following theorem.

THEOREM, STRONG INVARIANCE

$$\forall\, x : \mathcal{D}\,f\ .\ \forall\, n : \mathbb{N}\ .\ n \le \mu\,x \Rightarrow f\,x = f\,(g^n\,x) \tag{91}$$

PROOF. For arbitrary $x : \mathcal{D}\,f$, let $P : \mathbb{N} \to \mathbb{B}$ with $P\,n \equiv n \le \mu\,x \Rightarrow f\,x = f\,(g^n\,x)$.

We prove $\forall\, P$ by induction, that is, $P\,0$ and $\forall\, n : \mathbb{N}\ .\ P\,n \Rightarrow P\,(n+1)$.

$$\begin{array}{lll}
P\,0 \equiv & \langle \text{Definition } P \rangle & 0 \le \mu\,x \Rightarrow f\,x = f\,(g^0\,x) \\
\equiv & \langle g^0\,x = x, \text{refl.} = \rangle & 0 \le \mu\,x \Rightarrow 1 \\
\equiv & \langle p \Rightarrow 1 \equiv 1 \rangle & 1 \\
P\,n \equiv & \langle \text{Definition } P \rangle & n \le \mu\,x \Rightarrow f\,x = f\,(g^n\,x) \\
\Rightarrow & \langle n < m \Rightarrow n \le m \rangle & n < \mu\,x \Rightarrow f\,x = f\,(g^n\,x) \\
\equiv & \langle \text{Extract below} \rangle & n < \mu\,x \Rightarrow f\,x = f\,(g^{n+1}\,x) \\
\equiv & \langle n < m \equiv n+1 \le m \rangle & n+1 \le \mu\,x \Rightarrow f\,x = f\,(g^{n+1}\,x) \\
\equiv & \langle \text{Definition } P \rangle & P\,(n+1).
\end{array}$$

To reduce writing, we extracted the proof for $n < \mu\,x \Rightarrow f\,(g^n\,x) = f\,(g^{n+1}\,x)$:

$$\begin{array}{ll}
n < \mu\,x \Rightarrow \langle \text{Lemmata B,D below} \rangle & x \in \mathcal{D}\,g^n \wedge g^n\,x \in X_B \wedge g^n\,x \in X_Q \\
\Rightarrow \langle X_Q = \mathcal{D}\,f, \text{eq. (87)} \rangle & x \in \mathcal{D}\,g^n \wedge g^n\,x \in X_B \wedge f\,(g^n\,x) = f\,(g\,(g^n\,x)) \\
\Rightarrow \langle X_B \subseteq \mathcal{D}\,g, \text{definit.} \circ \rangle & f\,(g^n\,x) = f\,(g^{n+1}\,x).
\end{array}$$

Intuitively, some steps may appear too detailed, but only if one takes for granted certain properties of g^n, neglecting its definition. Verifying the properties involves a few subtle domain aspects collected in the following lemmata, whose proofs can be found in Boute [2004].

LEMMATA FOR COMPOSITION. *The following properties are successive refinements.*

A. $\forall x : X \,.\, \forall n : \mathbb{N} \,.\, x \in \mathcal{D}\, g^n \wedge g^n\, x \in X_B \Rightarrow x \in \mathcal{D}\, g^{n+1} \wedge g^{n+1}\, x \in X$
B. $\forall x : X \,.\, \forall n : \mathbb{N} \,.\, n < \mu\, x \Rightarrow x \in \mathcal{D}\, g^n \wedge g^n\, x \in X_B$
C. $\forall x : X \,.\, \forall n : \mathbb{N} \,.\, n \leq \mu\, x \Rightarrow x \in \mathcal{D}\, g^n \wedge g^n\, x \in X$
D. $\forall x : X_Q \,.\, \forall n : \mathbb{N} \,.\, n \leq \mu\, x \Rightarrow x \in \mathcal{D}\, g^n \wedge g^n\, x \in X_Q$

The following corollary now yields the obvious solution.

COROLLARY, SOLUTION OF THE RECURSION EQUATION. *The unique solution of* (84) *that terminates and satisfies* (85) *is* $f : X_Q \to \mathcal{R}\, h$ *with* $f\, x = h\,(g^{\mu\, x}\, x)$.

PROOF. Termination of $f\, x$ requires $x \in X_Q$. For any such x, we calculate $f\, x$:

$f\, x = \langle$Strong invariance$\rangle \quad f\,(g^{\mu\, x}\, x) \quad = \langle$Eq. (87) with $= B\,(g^{\mu\, x}\, x)\rangle \quad h\,(g^{\mu\, x}\, x)$.

Remark. Since the domain is fully defined by requiring termination, multiple/least fixpoints (as is customary for "partial" functions) need not be considered. This does not detract from fixpoint theory as essential background for any professional software engineer. Multiple views on the same topic always contribute to a better understanding. In fact, this minitheory is a good preamble to fixpoint theory.

6.4.4 *Practical Reasoning about Recursion: Invariants and Bound Functions.* While interesting theoretically, the explicit solution (especially calculating $\mu\, x$) is not convenient in practice. This is alleviated by developing a minitheory of *invariants* and *bound functions*, inspired by similarly named concepts in imperative programming, but too rarely (if at all) done in recursion theory.

6.4.4.1 *Recursion Invariants*

Definition, (*Invariant*). An *invariant* for (84) is a predicate $I : X \to \mathbb{B}$ satisfying $\forall x : \mathcal{D}\, f \,.\, B\, x \Rightarrow I\, x \Rightarrow I\,(g\, x)$. (92)

THEOREM. For I as defined, $\forall x : \mathcal{D}\, f \,.\, I\, x \Rightarrow \forall n : \mathbb{N} \,.\, n \leq \mu\, x \Rightarrow I\,(g^n\, x)$ (93)

PROOF. Given $x : \mathcal{D}\, f$ satisfying $I\, x$, let $P : \mathbb{N} \to \mathbb{B}$ with $P\, n \equiv n \leq \mu\, x \Rightarrow I\,(g^n\, x)$. We prove $\forall\, P$ by induction, that is, $P\, 0$ and $\forall n : \mathbb{N} \,.\, P\, n \Rightarrow P\,(n+1)$.

$$
\begin{array}{rcll}
P\, 0 & \equiv & \langle \text{Definition } P \rangle & 0 \leq \mu\, x \Rightarrow I\,(g^0\, x) \\
 & \equiv & \langle g^0\, x = x, I\, x \rangle & 0 \leq \mu\, x \Rightarrow 1 \\
 & \equiv & \langle p \Rightarrow 1 \equiv 1 \rangle & 1 \\
P\, n & \equiv & \langle \text{Definition } P \rangle & n \leq \mu\, x \Rightarrow I\,(g^n\, x) \\
 & \Rightarrow & \langle n < m \Rightarrow n \leq m \rangle & n < \mu\, x \Rightarrow I\,(g^n\, x) \\
 & \Rightarrow & \langle \text{Excerpt below} \rangle & n < \mu\, x \Rightarrow I\,(g^{n+1}\, x) \\
 & \equiv & \langle n < m \equiv n+1 \leq m \rangle & n+1 \leq \mu\, x \Rightarrow I\,(g^{n+1}\, x) \\
 & \equiv & \langle \text{Definition } P \rangle & P\,(n+1)
\end{array}
$$

To reduce writing, we excerpt the proof for $n < \mu\, x \Rightarrow I\,(g^n\, x) \Rightarrow I\,(g^{n+1}\, x)$:

$$
\begin{array}{rcll}
n < \mu\, x & \Rightarrow & \langle \text{Lemmata B, D} \rangle & x \in \mathcal{D}\, g^n \wedge g^n\, x \in X_Q \wedge g^n\, x \in X_B \\
 & \Rightarrow & \langle \mathrm{X}_Q = \mathcal{D}\, f, (92) \rangle & x \in \mathcal{D}\, g^n \wedge g^n\, x \in X_B \wedge (I\,(g^n\, x) \Rightarrow I\,(g\,(g^n\, x)) \\
 & \Rightarrow & \langle \mathrm{X}_B \subseteq \mathcal{D}\, g, def.\ \circ \rangle & I\,(g^n\, x) \Rightarrow I\,(g^{n+1}\, x)
\end{array}
$$

COROLLARY. *If I is an invariant for* (84) *where $h := id$ and P is a predicate on X satisfying $\forall x : \mathcal{D}\, f \,.\, I\, x \wedge \neg B\, x \Rightarrow P\, x$, then $\forall x : \mathcal{D}\, f \,.\, I\, x \Rightarrow P\,(f\, x)$.*

PROOF. For any $x : \mathcal{D}\, f \wedge I\, x$, clearly $\neg B\,(g^{\mu x}\, x)$, whereas (93) yields $I\,(g^{\mu x}\, x)$.

6.4.4.2 *Independent Bound Functions*

Definition. *An independent bound function for* (84) *is a function $\beta : X \to \mathbb{Z}$ satisfying (a) $\forall x : X \,.\, B\, x \Rightarrow \beta\,(g\, x) < \beta\, x$ and (b) $\forall x : X \,.\, B\, x \Rightarrow 0 < \beta\, x$* (94)

THEOREM, TERMINATION. *If* (84) *has an independent bound function β, then $\forall\, Q$.*

PROOF. Assume β is a function as stated. Since $\forall\, Q \equiv \forall x : X \,.\, \neg \forall n : \mathbb{N} \,.\, B\,(g^n\, x)$ by duality (26), we compute $\forall n : \mathbb{N} \,.\, B\,(g^n\, x)$ for arbitrary $x : X$.

$$
\begin{array}{lll}
\forall n : \mathbb{N} \,.\, B\,(g^n\, x) & & \\
\quad \equiv & \langle \text{Lemma A below} \rangle & \forall n : \mathbb{N} \,.\, B\,(g^n\, x) \wedge \beta\,(g^{n+1}\, x) < \beta\,(g^n\, x) \\
\quad \equiv & \langle m < n \equiv m \leq m - 1 \rangle & \forall n : \mathbb{N} \,.\, B\,(g^n\, x) \wedge \beta\,(g^{n+1}\, x) \leq \beta\,(g^n\, x) - 1 \\
\quad \equiv & \langle \text{Lemma B below} \rangle & \forall n : \mathbb{N} \,.\, B\,(g^n\, x) \wedge \beta\,(g^n\, x) \leq \beta\, x - n \\
\quad \Rightarrow & \langle \text{Inst. } n := 0, n := \beta\, x \rangle & B\,(g^0\, x) \wedge (\beta\, x \in \mathbb{N} \Rightarrow B\,(g^{\beta x}\, x) \wedge \beta\,(g^{\beta x}\, x) \leq 0) \\
\quad \Rightarrow & \langle \forall x : X \,.\, B\, x \Rightarrow 0 < \beta\, x \rangle & \beta\, x \in \mathbb{N} \wedge (\beta\, x \in \mathbb{N} \Rightarrow 0 < \beta\,(g^{\beta x}\, x) \leq 0) \\
\quad \Rightarrow & \langle \text{Modus ponens} \rangle & 0 < \beta\,(g^{\beta x}\, x) \leq 0 \\
\quad \equiv & \langle 0 < m \leq 0 \equiv 0 \rangle & 0.
\end{array}
$$

LEMMATA, TWO AUXILIARY PROPERTIES. *(proofs left to the reader)*

A. *If β is a bound function, $\forall (n : \mathbb{N} \,.\, B\,(g^n\, x)) \Rightarrow \forall n : \mathbb{N} \,.\, \beta\,(g^{n+1}\, x) < \beta\,(g^n\, x)$.*

B. *If $f : \mathbb{N} \to \mathbb{Z}$ satisfies $\forall n : \mathbb{N} \,.\, f\,(n+1) \leq f\, n - 1$, then $\forall n : \mathbb{N} \,.\, f\, n \leq f\, 0 - n$.*

Replacing 0 in (94(b)) by any integer constant yields an alternative formulation.

6.4.4.3 *Invariant-Bound Function Pairs.* Sometimes a stronger antecedent is convenient.

Definition. An *invariant-bound function pair* is a pair of functions $I : X \to \mathbb{B}$ and $\beta : X \to \mathbb{Z}$ satisfying (a) $\forall x : X \,.\, I\, x \wedge B\, x \Rightarrow I\,(g\, x) \wedge \beta\,(g\, x) < \beta\, x$
(b) $\forall x : X \,.\, I\, x \wedge B\, x \Rightarrow 0 < \beta\, x$ (95)

THEOREM, TERMINATION. *If* (84) *has an invariant-bound function pair I, β, then $\forall x : X \,.\, I\, x \Rightarrow Q\, x \wedge I\,(g^{\mu x}\, x)$.*

PROOF (OUTLINE, DETAILS LEFT TO THE READER). Let I, β be as stated and consider $x : X$ such that $I\, x$. Combining I and B in $C := I \;\widehat{\wedge}\; B$, prove $\exists n : \mathbb{N} \,.\, \neg C\,(g^n\, x)$. Letting $\kappa : X_I \to \mathbb{N}$ with $\kappa\, x = \mathop{\Uparrow} n : \mathbb{N} \mid \neg C\,(g^n\, x)$, prove $\forall n : \mathbb{N} \,.\, n \leq \kappa\, x \Rightarrow I\,(g^n\, x)$. From $I\,(g^{\kappa x}\, x)$ and $\neg C\,(g^{\kappa x}\, x)$ deduce $\neg B\,(g^{\kappa x}\, x)$ and finally $Q\, x$ and $\kappa\, x = \mu\, x$. □

Letting $I := X \bullet 1$ yields the termination theorem for independent bound functions.

6.4.5 *Application: Deriving Axiomatic Semantics for Loops.* As in (83), the denotational semantics of the loop $l :=$ `while` b `do` c `od` is given by the equation $\mathcal{C}\, l\, s = \mathcal{E}\, b\, s\, ?\, \mathcal{C}\, l\,(\mathcal{C}\, c\, s) \mid s$, which is an instance of $f\, x = B\, x\, ?\, f\,(g\, x)) \mid h\, x$

under the substitution $f, B, g, h := \mathcal{C}\,l, \mathcal{E}\,b, \mathcal{C}\,c, id$ and $X := S$. Hence, (88) yields the termination predicate $Q : S \to \mathbb{B}$ with $Q\,s = \exists\, n : \mathbb{N}\ .\ \neg(\mathcal{E}\,b)((\mathcal{C}\,c)^n\,s)$. Equivalently, the termination meaning function is $\mathcal{Q} : C \to S \to \mathbb{B}$ with $\mathcal{Q}\,l\,s = \exists\, n : \mathbb{N}\ .\ \neg(\mathcal{E}\,b)((\mathcal{C}\,c)^n\,s)$.

By the strong invariance theorem (91), the solution to the recursion equation is $\forall\, s : S_Q\ .\ \mathcal{C}\,l\,s = (\mathcal{C}\,c)^{\mu\,s}\,s$ where $\mu : S_Q \to \mathbb{N}$ with $\mu\,s = \mathop{\Uparrow} n : \mathbb{N} \mid \neg(\mathcal{E}\,b)((\mathcal{C}\,c)^n\,s)$. This directly leads to an axiomatic semantics as characterized by (82) and elaborated next.

Definitions, Invariants and Bound Expressions. Let $l := \texttt{while}\ b\ \texttt{do}\ c\ \texttt{od}$.

An *invariant* for l is an assertion i such that $[i \wedge b]c[i]$

An *independent bound expression* for l is an expression e such that

$b \Rightarrow 0 < e$ and $[b \wedge v = e]c[e < v]$ (where v is a rigid variable).

An *invariant/bound pair* for l is an assertion i and an expression e such that $i \wedge b \Rightarrow 0 < e$ and $[i \wedge b \wedge v = e]c[i \wedge e < v]$ (where v is a rigid variable).

THEOREM. *If i, e is an i/b pair for $l := \texttt{while}\ b\ \texttt{do}\ c\ \texttt{od}$, then* $[i]l[i \wedge \neg b]$. (96)

PROOF. For i, b as stated, $\models [\![i \wedge b \Rightarrow 0 < e]\!] \equiv \forall\, s : S\ .\ \mathcal{E}\,i\,s \wedge \mathcal{E}\,b\,s \Rightarrow \mathcal{E}\,e\,s > 0$ and

$$
\begin{array}{ll}
& [i \wedge b \wedge v = e]c[i \wedge e < v] \\
\equiv & \langle \text{Eq. (82)} \rangle\ \ \forall\, s : S\ .\ \forall\, v : D\ .\ \mathcal{E}\,i\,s \wedge \mathcal{E}\,b\,s \wedge v = \mathcal{E}\,e\,s \Rightarrow \mathcal{Q}\,c\,s \wedge \mathcal{E}\,e\,(\mathcal{C}\,c\,s) < v \\
\equiv & \langle \text{One-pt.} \rangle\ \ \forall\, s : S\ .\ \mathcal{E}\,i\,s \wedge \mathcal{E}\,b\,s \Rightarrow \mathcal{Q}\,c\,s \wedge \mathcal{E}\,e\,(\mathcal{C}\,c\,s) < \mathcal{E}\,e\,s.
\end{array}
$$

So $I, \beta := \mathcal{E}\,i, \mathcal{E}\,e$ satisfies (95), hence $\forall\, s : S\ .\ \mathcal{E}\,i\,s \Rightarrow \mathcal{Q}\,l\,s \wedge \mathcal{E}\,i\,((\mathcal{C}\,c)^{\mu\,s}\,s)$ by the termination theorem. Adding $\neg(\mathcal{E}\,b)((\mathcal{C}\,c)^{\mu\,s}\,s)$ and substituting $\mathcal{C}\,l\,s = (\mathcal{C}\,c)^{\mu\,s}\,s$ yields $\forall\, s : S\ .\ \mathcal{E}\,i\,s \Rightarrow \mathcal{Q}\,l\,s \wedge \mathcal{E}\,i\,(\mathcal{C}\,l\,s) \wedge \neg(\mathcal{E}\,b)(\mathcal{C}\,l\,s)$, which is $[i]l[i \wedge \neg b]$.

6.4.6 *A Concrete Example: Euclid's Algorithm.* This example is chosen not just because it is one of the first well-documented algorithms in history, but also because its analysis illustrates some interesting decisions regarding the design of auxiliary functions to embed it in the general model.

On $\mathbb{P} := \mathbb{N}_{>0}$, define $\text{—}|\text{—} : \mathbb{P}^2 \to \mathbb{B}$ with $d\,|\,a \equiv a/d \in \mathbb{P}$ ("d divides a"). Next, let $\text{—}\mathsf{cd}\text{—} : \mathbb{P} \times \mathsf{fam}\ \mathbb{P} \to \mathbb{B}$ with $d\ \mathsf{cd}\ f \equiv \forall\, i : \mathcal{D}\,x\ .\ d\,|\,f\,i$. The *greatest common divisor* relation is $\text{—}\mathsf{isgcd}\text{—} : \mathbb{P} \times \mathsf{fam}\ \mathbb{P} \to \mathbb{B}$ with $d\ \mathsf{isgcd}\ f \equiv d\ \mathsf{cd}\ f\ \wedge\ \forall\, a : \mathbb{P}\ .\ a\ \mathsf{cd}\ f \Rightarrow a \leq d$. The greatest common divisor function is *specified* by $\mathsf{Gcd} : \mathsf{fam}\ \mathbb{P} \to \mathbb{P}$ with $\mathsf{Gcd}\ f\ \mathsf{isgcd}\ f$, but here we shall restrict it to pairs, that is:

$$\textbf{spec}\ \mathsf{Gcd} : \mathbb{P}^2 \to \mathbb{P}\ \textbf{with}\ \mathsf{Gcd}\,(x, y)\,\mathsf{isgcd}\,(x, y).$$

Existence and uniqueness can be proven directly from the definition, but also emerges as a fringe benefit from analyzing Euclid's algorithm.

(a) Consider the recursive *functional program* or recursion equation

$$\textbf{def}\ G : \mathbb{P}^2_Q \to \mathbb{P}\ \textbf{with}\ G\,(x, y) = (x = y)\,?\,x \mid (x < y)\,?\,G\,(x, y - x) \mid G\,(y, x - y),$$

where $Q : \mathbb{P}^2 \to \mathbb{B}$ is the termination predicate (to be determined) for G. The image definition can be written in the general form (84) as

$$G\,(x, y)\ =\ (\neq)(x, y)\,?\,G\,(g\,(x, y)) \mid (x, y)\,0$$

where $g : \mathbb{P}^2_{\neq} \to \mathbb{P}^2$ with $g\,(x, y) = (x < y)\,?\,(x, y - x) \mid (y, x - y)$.

One can show that $\beta : \mathbb{P}^2 \to \mathbb{P}$ with $\beta\,(x\ y) = (x < y)\,?\ y \nmid x$ is an independent bound function, that is, $\forall\, p : \mathbb{P}^2 \,.\, (\neq)\, p \Rightarrow \beta\,(g\ p) < \beta\, p$, which ensures $\forall\, Q$ or $Q = \mathbb{P}^2 \bullet 1$. Hence, the solution to the recursion equation is

$$G : \mathbb{P}^2 \to \mathbb{P} \ \text{ with } \ G\,(x, y) = g^m\,(x, y)\,0 \ \textbf{where}\ m := \bigwedge n : \mathbb{N} \ .\ (=)(g^n\,(x, y)).$$

From the specification of Gcd, one can deduce

$$(x = y \Rightarrow \mathsf{Gcd}\,(x, y) = x) \ \wedge \ (x \neq y \Rightarrow \mathsf{Gcd}\,(g\,(x, y)) = \mathsf{Gcd}\,(x, y)),$$

yielding the same recursion equation as the one for G, which has a unique solution as given. One can also show that G indeed satisfies the specification.

(b) Consider now the *imperative program* loop

$$l := \texttt{while x /= y do if x < y then y := y - x else x := x - y fi od}.$$

The state space is $S := \{\underline{\mathrm{x}}, \underline{\mathrm{y}}\} \to \mathbb{P}$. With our earlier convention, we can write elements of S as $\langle p \rangle$ such that $\langle p \rangle\, \underline{\mathrm{x}} = p\,0$ and $\langle p \rangle\, \underline{\mathrm{y}} = p\,1$, where $p : \mathbb{P}^2$.

With b and c as "rigid variables" or "ghost variables", we introduce the invariant and bound expression

$$\begin{aligned} i \ &:= \ [\![0 < \mathrm{x} \wedge 0 < \mathrm{y} \wedge \mathsf{Gcd}\,(\mathrm{x}, \mathrm{y}) = \mathsf{Gcd}\,(b, c)]\!] \\ e \ &:= \ [\![(\mathrm{x} < \mathrm{y})\,?\,\mathrm{y} \nmid \mathrm{x}]\!]. \end{aligned}$$

Take the precondition $a := [\![0 < b \wedge 0 < c \wedge \mathrm{x} = b \wedge \mathrm{y} = c]\!]$, which clearly implies i. Hence, by theorem (96), $[a]l[i \wedge \mathrm{x} = \mathrm{y}]$. The postcondition $i \wedge \mathrm{x} = \mathrm{y}$ implies $\mathsf{Gcd}\,(b, c) = \mathsf{Gcd}\,(\mathrm{x}, \mathrm{y}) = \mathrm{x} = \mathrm{y}$.

This example also illustrates how the many technical details that are necessary for the derivation and justification of axiomatic semantics culminating in theorem (96) become hidden in the application of axiomatic semantics to concrete problems.

7. FINAL REMARKS AND CONCLUSION

7.1 Mechanization, Calculational Reasoning and Proof Style

Although the formalism presented is not (yet) automated, it can make an important contribution to the more effective use of automated tools in the long term.

Indeed, in classical engineering areas, mechanized tools such as Maple, Mathematica, Mathcad, MATLAB and Simulink have been easily adopted and are widely used by practicing engineers. This is because they are based on algebra and analysis, where the formalisms have evolved for human use (the hallmark being formal calculation by hand), are part of every engineering curriculum, and are routinely used in applications. Experience shows that a 50-year old engineering calculus text is a good basis for immediate and problem-free use of such tools.

The situation is quite different for logic-oriented tools. Current theorem provers (like PVS [Rushby et al. 1998], HOL and variants [Harrison 2000; Paulson 2001]) and model checkers (like SPIN [Holzmann 2003]) are not so well adopted, and practical use typically relies on experts. This is because they implement logic formalisms designed for automation, not for calculation by

humans. Support for calculational logic is still very rare [Manolios and Moore 2001], although gradually emerging [Harrison 2000].

Basically, logic has skipped a crucial evolutionary step: unlike classical algebra and calculus, it never had the opportunity to develop into a calculational tool for use by humans [Dijkstra 2000; Gries 1996a] before attention shifted to automation (predating the computer era).

Formalisms as presented in this article can provide the missing link, ultimately facilitate the adoption of mechanized tools, and in the meantime address issues that are still beyond current tools (which in the best case can only detect design errors).

Examples of such issues are: offering multiple views or solutions, and proof style. Proofs should be elegant, that is: concise, clear, not hiding difficult or essential steps, and appealing to an aesthetic sense. The first three characteristics are ensured by the calculational approach, the last requires considerable effort on the part of the writer [Lamport 1993]. Gries and Schneider [1993] emphasizes the need for repeated polishing or even redesign before a proof can be considered sufficiently "finished". Having proofs read and criticized by others is most helpful. The proofs in this article are somewhat uneven in this respect; they even include first attempts that have not been discussed with others. Readers are therefore kindly invited to contribute to this process of continuing improvement.

It is also clear that the concerns in this article are quite different from classical formal logic: they certainly are not the usual foundational ones about building mathematics from scratch, but practical, about providing a framework in which mathematical concepts and reasoning can be fitted and equipped with formal calculation rules.

By the same token, if we presented a few examples that seem not directly related to programming or languages as a topic, the great similarity in the calculations is meant to show how much our predicate calculus and generic functionals erases distinctions and rapproaches various mathematical disciplines methodologically.

7.2 Conclusion

The following has been demonstrated:

—The effectiveness of the formalism in (re)synthesizing common conventions, while also generating new calculational properties, covering new areas, and reducing unnecessary diversity and fragmentation. This is also crucial in education.

—The syntactic simplicity obtained by functional semantics and full orthogonality.

—Point-free and point-wise styles and formal rules for transition between them.

—The greater generality of elastic operators with respect to other generalizations of quantification.

—A formulation of predicate calculus that is simpler than ordinary summation, yet provides a very rich collection of calculation laws that enables practical use.

—The unifying power of generic functionals by encapsulating calculational properties in a domain-independent way.

—The wide range of applications in general, and in programming language design and semantics in particular.

The functional predicate calculus gives working mathematicians and engineers the opportunity to make formal manipulation of quantified expressions as routine as the familiar differential and integral calculus. Together with the generic functionals, it provides the glue between "continuous" and discrete applied mathematics in general, and between classical engineering and software engineering in particular.

REFERENCES

AARTS, C., BACKHOUSE, R., HOOGENDIJK, P., VOERMANS, E., AND VAN DER WOUDE, J. 1992. *A Relational Theory of Data Types*. Lecture notes, T. U. Eindhoven.

ALMSTRUM, V. L. 1996. Investigating student difficulties with mathematical logic. In *Teaching and Learning Formal Methods*, C. Neville Dean and Michael G. Hinchey, Eds. Academic Press, Orlando, FL, pp. 131–160.

ALUR, R., HENZINGER, T. A., AND SONTAG, E. D., EDS. 1996. *Hybrid Systems III*. Lecture Notes in Computer Science, vol. 1066. Springer-Verlag, Berlin, Heidelberg.

BACKUS, J. 1978. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Commun. ACM 21*, 8, (Aug.) 613–641.

BARENDREGT, H. P. 1984. *The Lambda Calculus, Its Syntax and Semantics*. North-Holland, Amsterdam, The Netherlands.

BIRD, R. 1998. *Introduction to Functional Programming using Haskell*. Prentice-Hall International Series in Computer Science, London, England.

BISHOP, R. H. 2001. *LabVIEW Student Edition*, Prentice-Hall, Englewood Cliffs, N.J.

BOITEN, E. AND MÖLLER, B. 2002. In *Proceedings of the 6th International Conference on Mathematics of Program Construction* (Conference announcement). Dagstuhl. `http://www.cs.kent.ac.uk/events/conf/2002/mpc2002`.

BOUTE, R. T. 1982. On the requirements for dynamic software modification. In *MICROSYSTEMS: Architecture, Integration and Use*, van Spronsen, C. J. and L. Richter, Eds., North-Holland, Amsterdam, The Netherlands, pp. 259–271.

BOUTE, R. T. 1990. A heretical view on type embedding. *ACM SIGPLAN Notices 25*, 11 (Jan.), 22–28.

BOUTE, R. T. 1991. Declarative languages—Still a long way to go. In *Computer Hardware Description Languages and their Applications*. Dominique Borrione and Ronald Waxman, Eds. North-Holland, Amsterdam, The Netherlands, pp. 185–212.

BOUTE, R. T. 1992. The Euclidean definition of the functions div and mod. *ACM Trans. Prog. Lang. Syst. 14*, 2 (Apr.), 127–144.

BOUTE, R. T. 1993a. *Funmath Illustrated: A Declarative Formalism and Application Examples*. Declarative Systems Series No. 1, Computing Science Institute, University of Nijmegen.

BOUTE, R. T. 1993b. Fundamentals of hardware description languages and declarative languages. In *Fundamentals and Standards in Hardware Description Languages*. Jean P. Mermet, Ed. Kluwer Academic Publishers, pp. 3–38.

BOUTE, R. T. 2000. Supertotal function definition in mathematics and software engineering. *IEEE Trans. Softw. Eng. 26*, 7 (July), pp. 662–672.

BOUTE, R. T. 2002. *Functional Mathematics: A Unifying Declarative and Calculational Approach to Systems, Circuits and Programs—Part I: Basic Mathematics*. Course text, Ghent University.

BOUTE, R. T. 2003. Concrete generic functionals: Principles, design and applications. In *Generic Programming*, Jeremy Gibbons, Johan Jeuring, Eds., Kluwer, pp. 89–119.

BOUTE, R. 2004. Functional declarative language design and predicate calculus: A practical approach. Tech. Rep. TR-B2004-03. INTEC, Ghent Univ.

BRYANT, R. E. 1992. Symbolic Boolean manipulation with ordered binary-decision diagrams. *Comput. Surv. 24*, 3, (Sept.), 293–318.

BUCK, J., HA, S., LEE, E. A., AND MESSERSCHMITT, D. G. 1994. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *Int. J. Comput. Simul. (spec. issue on Simulation Software Development)*, (Jan.)

CARSON, R. S. 1990. *Radio Communications Concepts: Analog*. Wiley, New York.

COHEN, E. 1990. *Programming in the 1990s*. Springer-Verlag, Berlin, Germany.

CORI, R. AND LASCAR, D. 2000. *Mathematical Logic: A Course with Exercises, Part I*. Oxford University Press, Oxford, England.

DAVENPORT, J. H. 2000. A small OpenMath type system. *SIGSAM Bull. 34*, 2, (Jun.), 16–26.

DEAN, C. N. AND HINCHEY, M. G. 1996. *Teaching and Learning Formal Methods*. Academic Press, London, England, 197–243.

DIJKSTRA, E. W. 1992. On the economy of doing mathematics. *EWD1130*, http://www.cs.utexas.edu/users/EWD/.

DIJKSTRA, E. W. 1996a. Beware of the empty range. *EWD1247* (Sep.), http://www.cs.utexas.edu/users/EWD/.

DIJKSTRA, E. W. 1996b. Andrew's challenge once more. *EWD1249* (Oct.), http://www.cs.utexas.edu/users/EWD/.

DIJKSTRA, E. W. 2000. Under the spell of Leibniz's dream. *EWD1298* (Apr.), http://www.cs.utexas.edu/users/EWD/.

DIJKSTRA, E. W. AND SCHOLTEN, C. S. 1990. *Predicate Calculus and Program Semantics*. Springer-Verlag, Berlin, Germany.

FORSTER, T. E. 1992. *Set Theory with a Universal Set*. Clarendon Press, Oxford, England.

GIERZ, G., HOFMANN, K. H., KEIMEL, K., LAWSON, J. D., MISLOVE, M., AND SCOTT, D. S. 1980. *A Compendium of Discrete Lattices*. Springer-Verlag, Berlin, Germany.

GRAHAM, R. L., KNUTH, D. E., AND PATASHNIK, O. 1994. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, Reading, MA.

GRIES, D. 1996a. The need for education in useful formal logic. *IEEE Comput. 29*, 4, (Apr.), 29–30.

GRIES, D. 1996b. A calculational proof of Andrew's challenge. Technical Note, `http://webster.cs.uga.edu/~gries/Papers/andrews.pdf`.

GRIES, D. AND SCHNEIDER, F. B. 1993. *A Logical Approach to Discrete Math*. Springer-Verlag, New York.

HALMOS, P. AND GIVANT, S. 1998. *Logic as Algebra*. Math. Assoc. Amer.

HARRISON, J. 2000. *The HOL Light Manual*. University of Cambridge Computer Laboratory, `http://www.cl.cam.ac.uk/users/jrh/hol-light/manual-1.1.pdf`.

HEHNER, E. C. R. 1996. Boolean formalism and explanations. Invited talk, *AMAST*, (Munich, Germany, (July), Text at `http://www.cs.toronto.edu/~hehner/BFE.pdf`.

HOLZMANN, G. 2003. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, Reading, Mass.

HUDAK, P., PETERSON, J., AND FASEL, J. H. 1999. *A Gentle Introduction to Haskell 98* (Oct.). `http://www.haskell.org/tutorial/`

HUDAK, P., COURTNEY, A., NILSSON, H., AND PETERSON, J. 2003. Arrows, robots and functional reactive programming. In *Advanced Functional Programming*, Johan Jeuring and Simon Peyton Jones, Eds., Lecture Notes in Computer Science, vol. 2638. Springer-Verlag, New York, haskell.CS.Yale.edu/Yampa/AFPLectureNotes.pdf.

IEEE 1994. *IEEE Standard VHDL Language, Reference Manual*. IEEE, New York.

ILLINGWORTH, V., GLASER, E. L., AND PYLE, I. C. 1989. *Dictionary of Computing*, Oxford University Press, Oxford, England.

JACKSON, P. B. 1992. NUPRL and its use in circuit design. In *Theorem Provers in Circuit Design*. Victoria Stavridou, Tom Melham, and Raymond T. Boute, Eds., North Holland, Amsterdam, The Netherlands, pp. 311–336.

JENSEN, K. AND WIRTH, N. 1978. *PASCAL User Manual and Report*. Springer-Verlag, New York.

JOHNSON-LAIRD, P. N. 2000. (example problems in the psychological study of human reasoning), `http://www.princeton.edu/~psych/PsychSite/fac_phil.html`.

LANG, S. 1983. *Undergraduate Analysis*. Springer-Verlag, Berlin, Germany.

LAMPORT, L. AND PAULSON, L. C. 1997. Should your specification language be typed? SRC Research Report 147, Digital Equipment Corporation (May).

LAMPORT, L. 1993. How to write a proof. DEC SRC Research Report. http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-94.html.

LOECKX, J. AND SIEBER, K. 1984. *The Foundations of Program Verification*. Wiley-Teubner, New York.

MANOLIOS, P. AND MOORE, J. S. 2001. On the desirability of mechanizing calculational proofs. *Inf. Proc. Lett., 77*, 2–4, 173–179.

MENDELSON, E. 1987. *Introduction to Mathematical Logic*, 3rd. ed. Wadsworth & Brooks/Cole.

MEYER, B. 1991. *Introduction to the Theory of Programming Languages*. Prentice Hall, New York.

OSTROFF, J. S. Rationale for restructuring the logic and discrete math (MATH1090/2090) courses for CS students. York University, http://www.cs.yorku.ca/~logicE/curriculum/logic_discrete_maths.html.

PAGE, R. 2000. *BESEME: Better Software Engineering through Mathematics Education*, project presentation http://www.cs.ou.edu/~beseme/besemePres.pdf

PARNAS, D. L. 1993. Predicate logic for software engineering. *IEEE Trans. SWE 19*, 9, (Sept.), 856–862.

PAULSON, L. C. 2001. *Introduction to Isabelle*, Cambridge Computer Laboratory. http://www.cl.cam.ac.uk/Research/HVG/Isabelle/dist/docs.html.

PUGH, W. 1994. Counting solutions to Presburger formulas: How and why. *ACM SIGPLAN Notices 29*, 6, (June), 121–122.

RECHENBERG, P. 1990. Programming languages as thought models. *Struct. Prog. 11*, 105–115.

REYNOLDS, J. C. 1980. Using category theory to design implicit conversions and generic operators. In *Semantics-Directed Compiler Generation*. Neil D. Jones, Ed., Lecture Notes in Computer Science, vol. 94, Springer-Verlag, Berlin, Germany, pp. 261–288.

ROBERTS, R. A. AND MULLIS, C. T. 1987. *Digital Signal Processing*. Addison-Wesley, Reading, MA.

RUSHBY, J., OWRE, S., AND SHANKAR, N. 1998. Subtypes for Specifications: Predicate subtyping in PVS. *Trans. Softw. Eng. 24*, 9, (Sept.), 709–720.

SCHIEDER, B. AND BROY, M. 1999. Adapting calculational logic to the undefined. *Comput. J. 42*, 2, 73–81.

SPIVEY, J. M. 1989. *The Z notation: A Reference Manual*. Prentice-Hall, Englewood, Cliffs, N. J.

STOY, J. E. 1977. *Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory*. The MIT Press, Cambridge, Mass.

TARSKI, A. AND GIVANT, S. 1987. *A Formalization of Set Theory without Variables*. American Mathematical Society.

TAYLOR, P. 2000. *Practical Foundations of Mathematics* (second printing), No. 59 in Cambridge Studies in Advanced Mathematics, Cambridge University Press, http://www.dcs.qmul.ac.uk/~pt/Practical_Foundations/html/s10.html

TENNENT, R. D. 1991. *Semantics of Programming Languages*. Prentice-Hall, Englewood Cliffs, NJ.

VAANDRAGER, F. W. AND VAN SCHUPPEN, J. H. EDS. 1999. *Hybrid Systems: Computation and Control*. Lecture Notes in Computer Science, vol. 1569. Springer-Verlag, Berlin Heidelberg, Germany.

VAN DEN BEUKEN, F. 1997. A functional approach to syntax and typing. Ph.D. dissertation. School of Mathematics and Informatics, University of Nijmegen.

VAN THIENEN, H. 1994. It's about time—using funmath for the specification and analysis of discrete dynamic systems. Ph.D. dissertation. Department of Informatics, University of Nijmegen.

WECHLER, W. 1987. *Universal Algebra for Computer Scientists*. Springer-Verlag, New York.

WINSKEL, G. 1993. *The Formal Semantics of Programming Languages: An Introduction*. The MIT Press, Cambridge, MA.

WOLFRAM, S. 1996. *The Mathematica book, 3rd. ed.* Cambridge University Press, Cambridge, MA.

ZAMFIRESCU, A. 1993. Logic and arithmetic in hardware description languages. In *Fundamentals and Standards in Hardware Description Languages*, J. P. Mermet, Ed. NATO ASI Series E, Vol. 249. Kluwer, Dordrecht, pp. 79–107.

Received July 2003; revised July 2004; accepted September 2004